**Trabajo 2**

**Redes neuronales convolucionales**

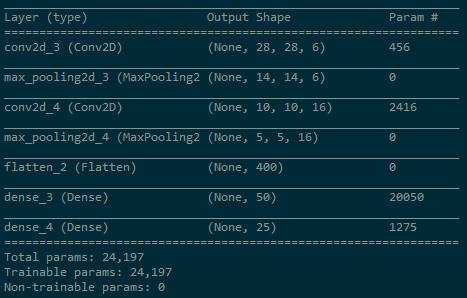
**Ignacio Vellido Expósito**

**Nota:** Se muestran los valores de test en cada gráfica, pero las mejoras no se valoran en base ellos, ya que sino

estaríamos filtrando información indirectamente en el conjunto de entrenamiento.

Apartado 1

La arquitectura del modelo se construye en base al guion de la práctica.



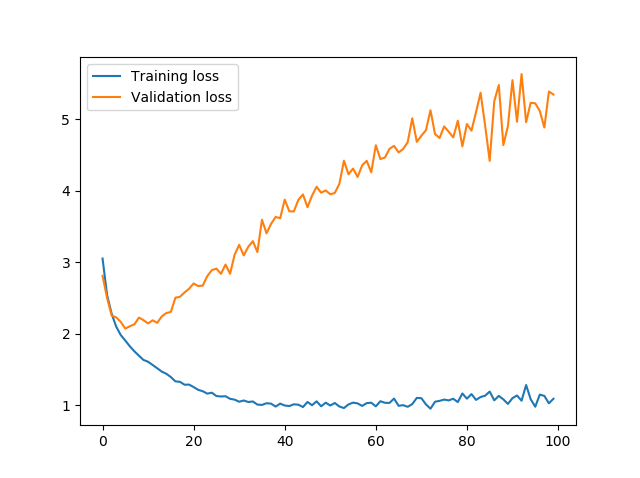
Puesto que estamos tratando con un problema de clasificación de múltiples clases, se le añade una activación *softmax* a la última capa, de manera que el modelo devuelva una probabilidad para cada clase.

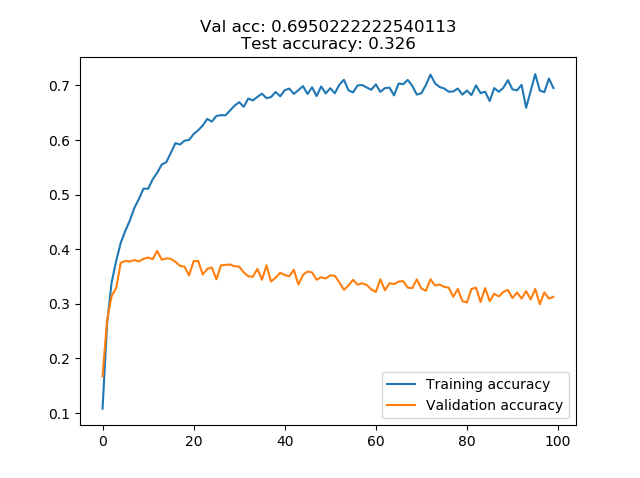
Para poder tener un valor de *accuracy* del que partir a la hora de hacer mejoras, se determinan los hiperparámetros y se entrena el modelo:

* Optimizador: Como optimizador se utiliza SGD, utilizando inicialmente los parámetros del documento de introducción a Keras.
* Batch-size: Al contar con un número suficientemente grande de datos podemos usar un tamaño de *batch* medianamente grande. En este caso se parte de 32.
* Epochs: El mejor número de épocas se decide experimentalmente:

**Nota:** El valor de *accuracy* mostrado en las dos primeras gráficas es erróneo, corresponde al de training.

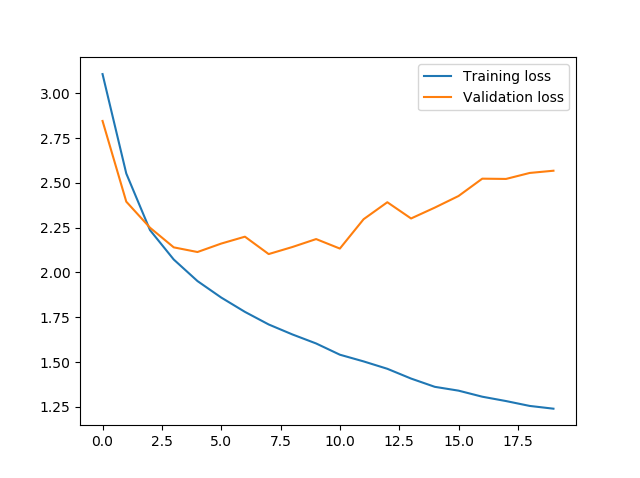
Primero se lanza con un número grande, en este caso de 100, y así poder precisar en base a la gráfica.

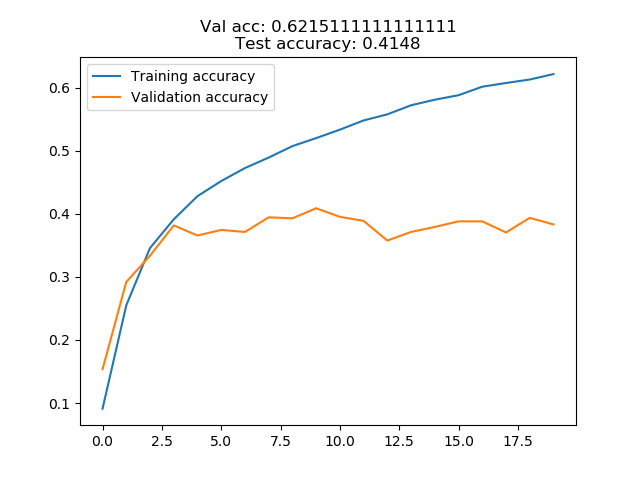




Vemos que el número ha sido exagerado, se ha producido demasiado *overfitting*.

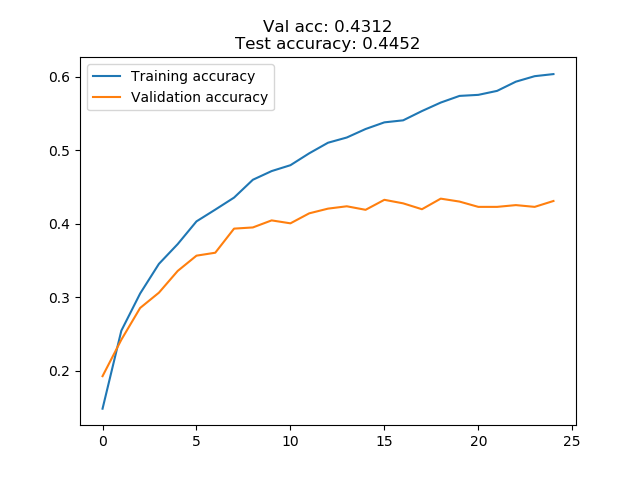
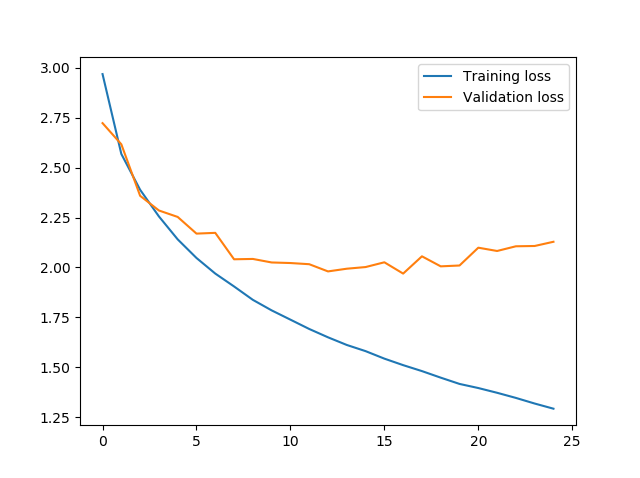
La gráfica muestra que entre 10 y 20 se comienza a dar el sobreajuste, así que se vuelve a repetir para 20.



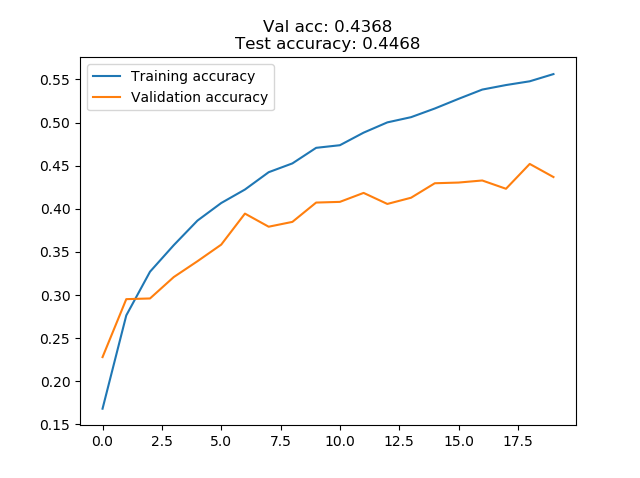
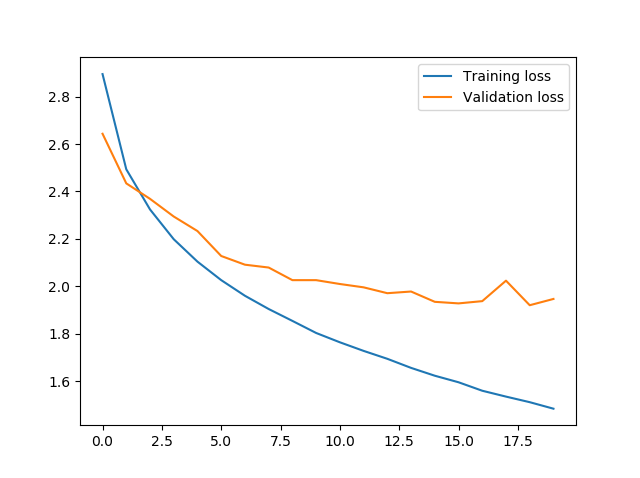


Aquí se aprecia que el sobreajuste comienza demasiado pronto (en torno a las 3 épocas), parece que la red aprende demasiado rápido. Se prueba con otro optimizador (RMSprop) y 25 épocas.

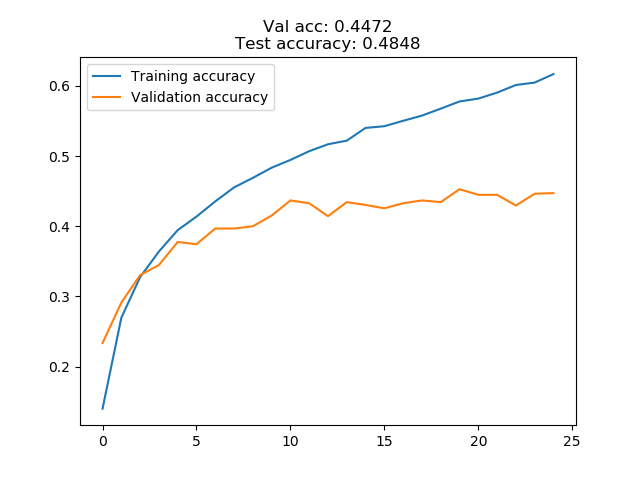
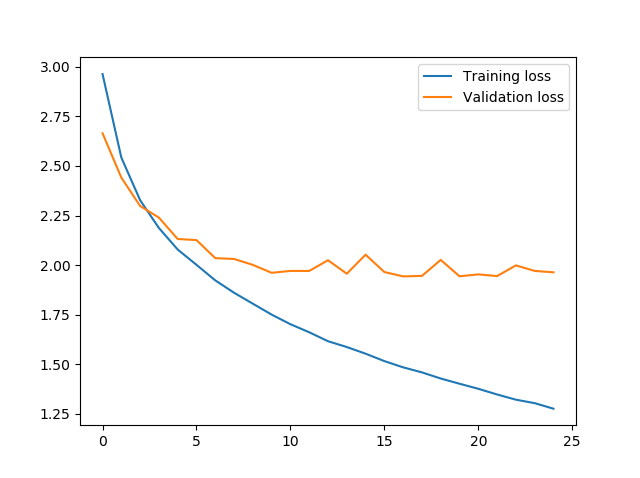
Se elige RMSprop porque controla bien el momentum de los gradientes y es posible que evite que se aprenda demasiado rápido.

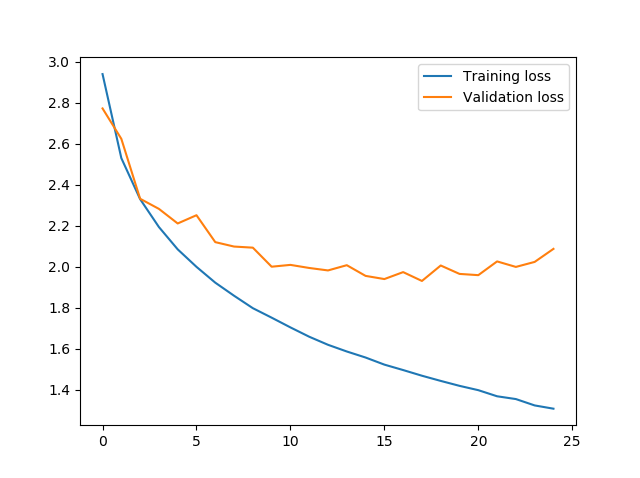


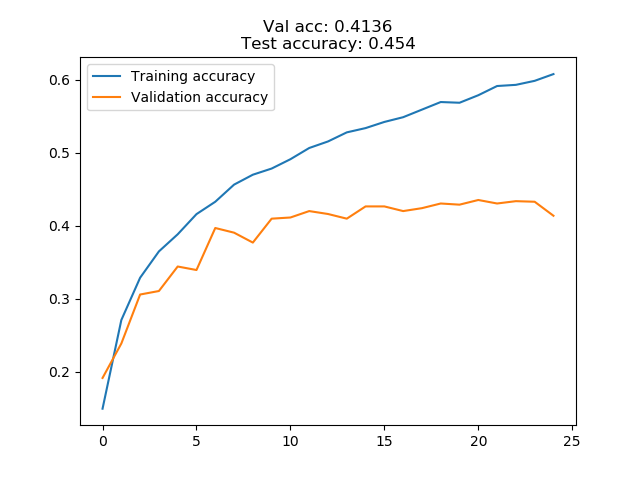
Se prueba de nuevo añadiendo un mayor número de épocas 20:



Viendo que este parece ser el número correcto, se lanzan varias iteraciones con ese valor para decidir el *accuracy* del modelo:







Se hace la media y se toma como valor de *accuracy* a mejorar: **0.4306**

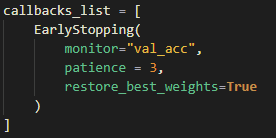
La manera óptima de determinar el potencial real del modelo habría sido una búsqueda profunda de todos los hiperparámetros, pero ya que esa estrategia consume demasiado tiempo se deja como está.

Apartado 2

Early-stopping

Antes que nada, aplicar una técnica de *early-stopping* se puede considerar como una mejora sin (aparentes) puntos negativos, ya que solo estamos conservando la red en su mejor momento del entrenamiento.

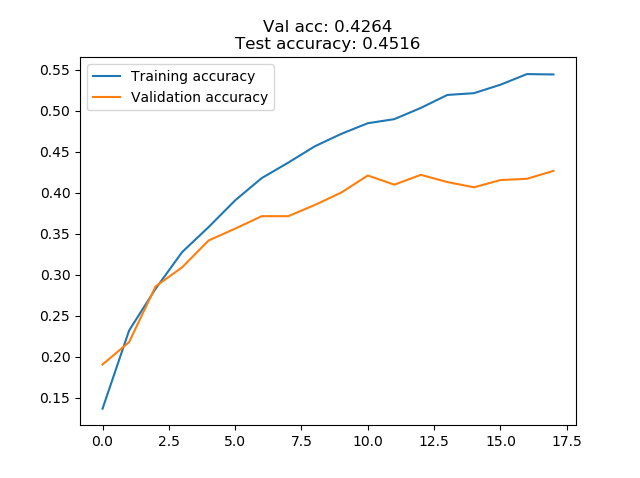
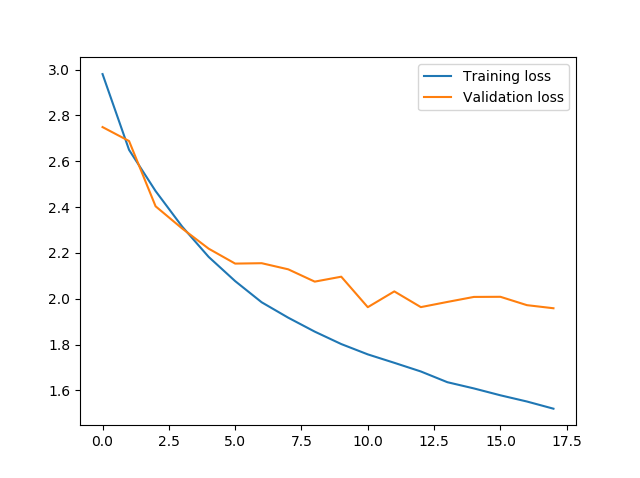
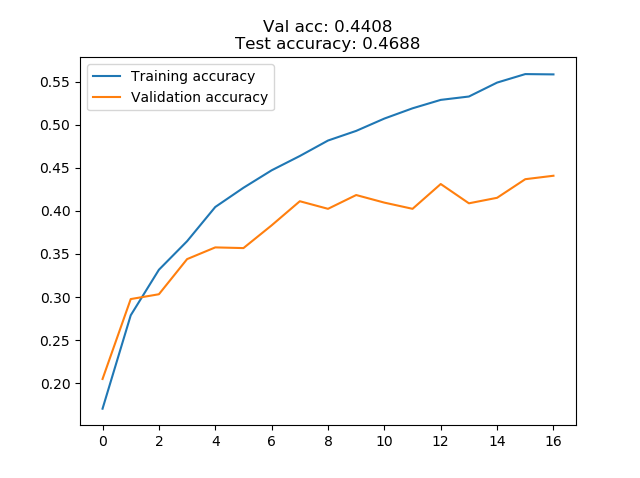
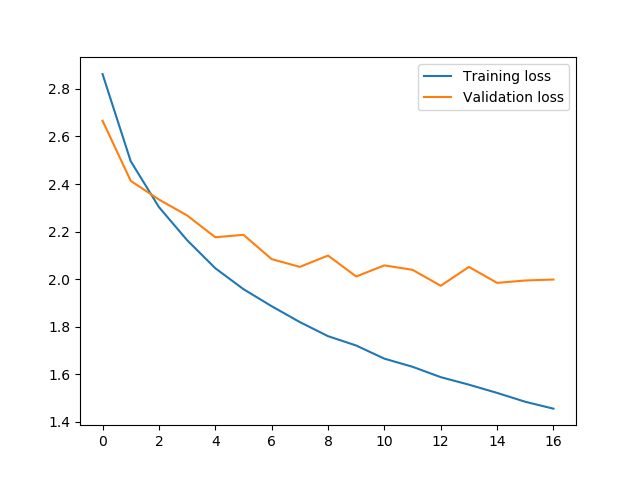
Es necesario por tanto implementar un callback para monitorizar la *accuracy* (es un mejor parámetro que la función de pérdida pues es la medida que estamos usando para evaluar el modelo).



Se pone *patience* a 3, indicando que solo se permite que el modelo empeore dos veces, si tras esa vez no se recupera se detendrá el entrenamiento.

Es posible que tras modificar la arquitectura sea necesario aumentar este valor para evitar que caiga en máximos locales. En este caso, viendo las gráficas del apartado anterior parece conveniente dejarlo a este valor. Además, viendo que fluctúa con facilidad es posible que se empeore si dejamos que entrene demasiado.

Una vez implementado, se realizan algunas pruebas:



Y esto nos afirma que nuestro mejor número de épocas era de 17/18 y el nuevo *accuracy* a superar es: **0.4336**

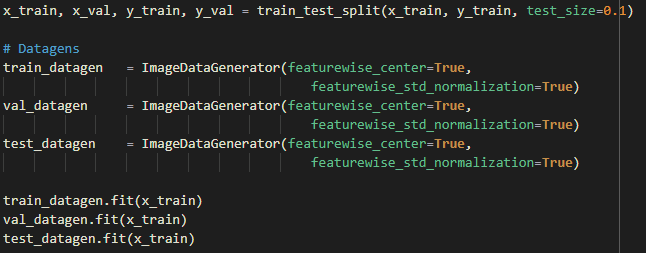
Normalización de datos

El tener diferentes rangos en los valores hace que las redes neuronales funcionen peor, ya que a la hora de realizar *backpropagation* hace que los gradientes alcancen valores dispares y no fluyan correctamente.

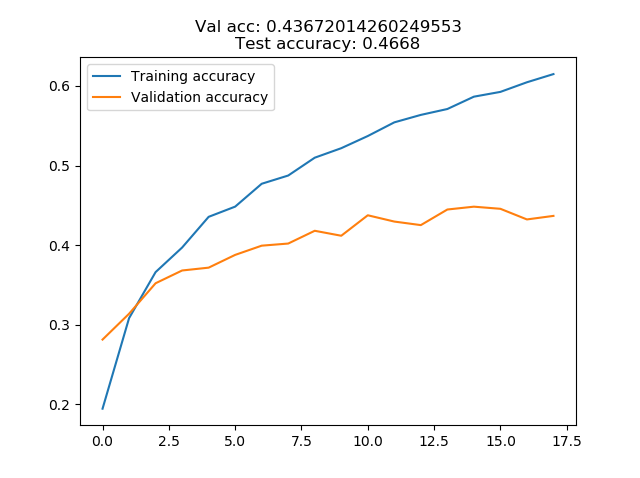
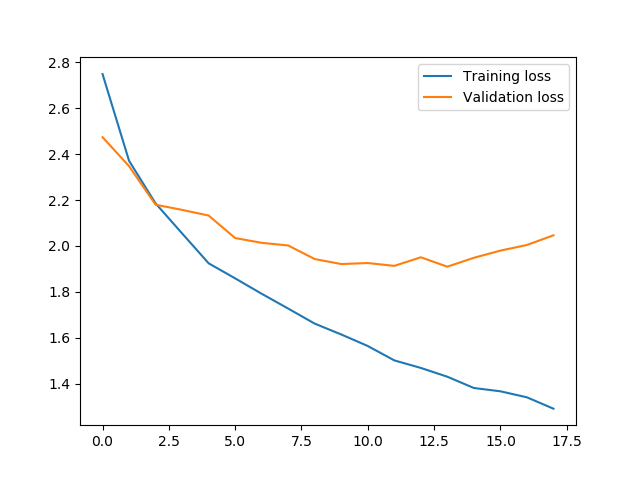
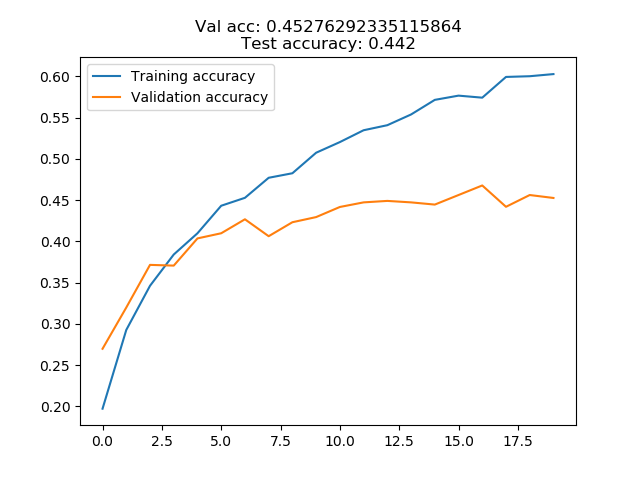
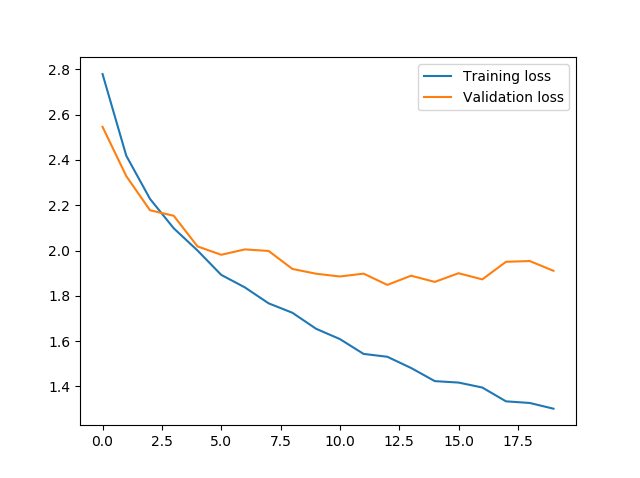
Se considera buena idea por tanto normalizar los datos de entrada con media cero y desviación típica uno.

Para esto se separa previamente un 10% del conjunto de entrenamiento y se crean los generadores necesarios.

Se utilizan dos *ImageGenerator* de cara a los siguientes apartados, pues el conjunto de validación no debe contener *data-augmentation*.



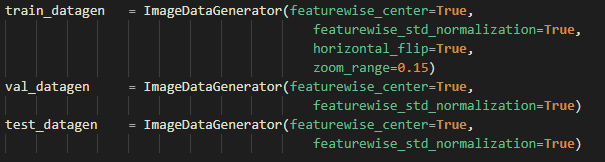
Es importante que todos los conjuntos reciban el mismo preprocesado, por ello tanto el conjunto de test como el de validación se ajustan sobre el conjunto de entrenamiento.



Aunque pequeña, se aprecia una mejora, y el nuevo valor de *accuracy* es: **0.4438**

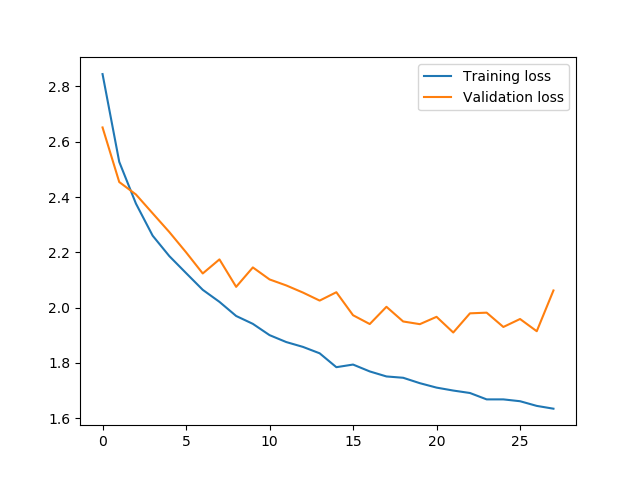
Aumento de datos

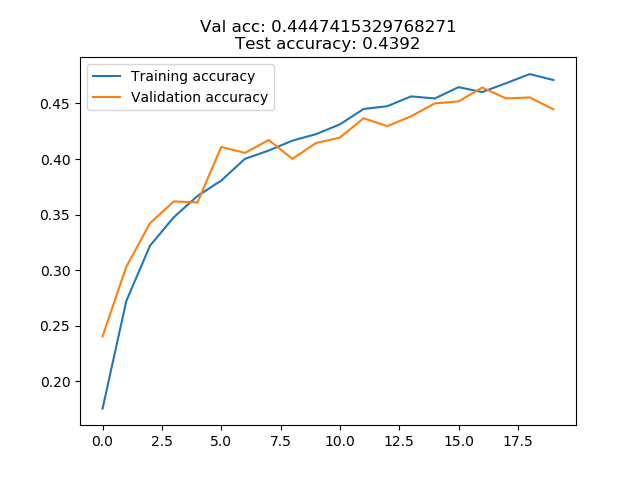
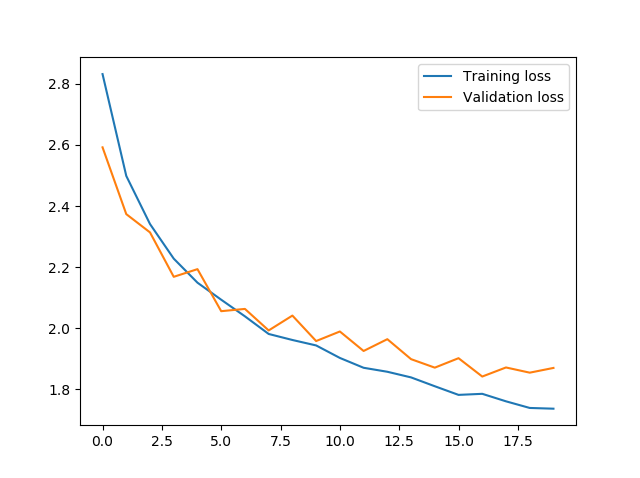
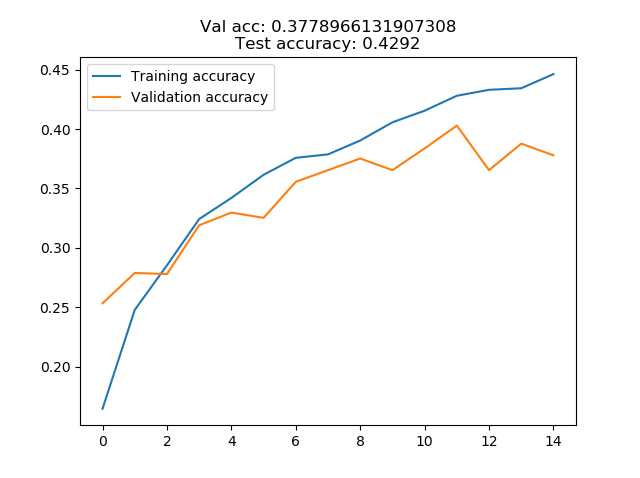
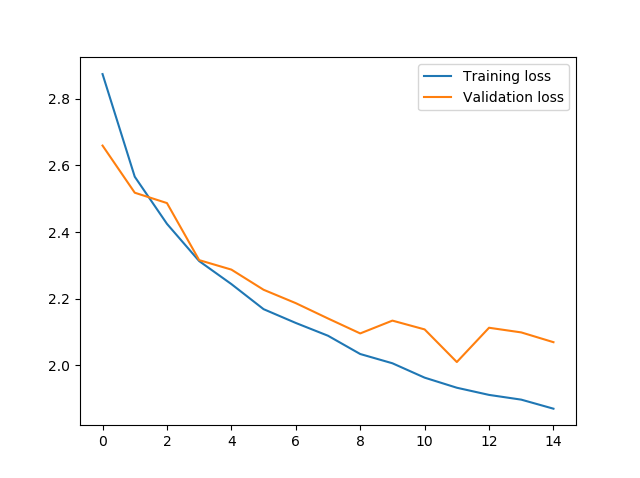
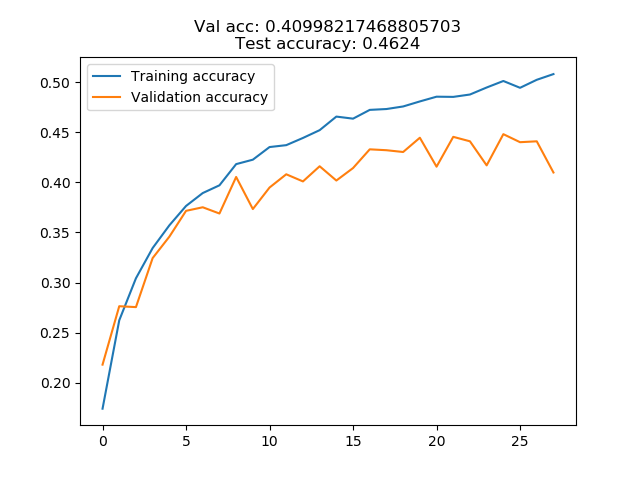
Una técnica para evitar el *overfitting* es el aumento del conjunto de datos, de manera que el modelo sea capaz de generalizar de mejor manera. Al estar trabajando con imágenes, existe una serie de transformaciones que podemos aplicar para aumentar este conjunto. La contra de esta idea es que hace que el entrenamiento sea mucho más lento.



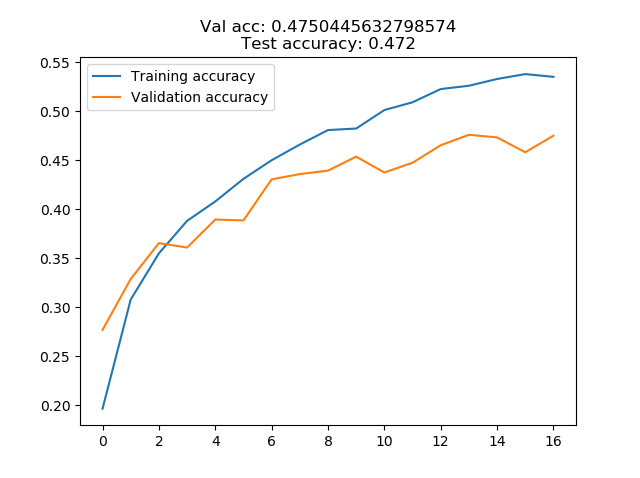
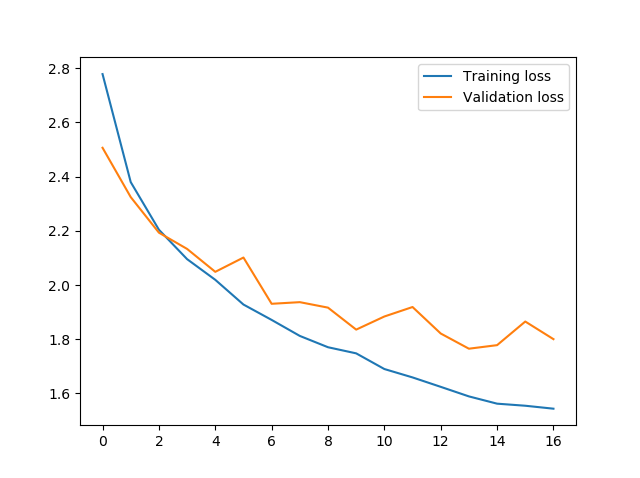
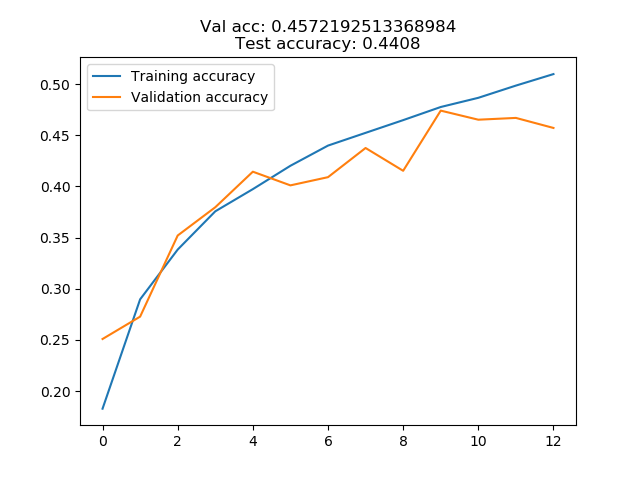
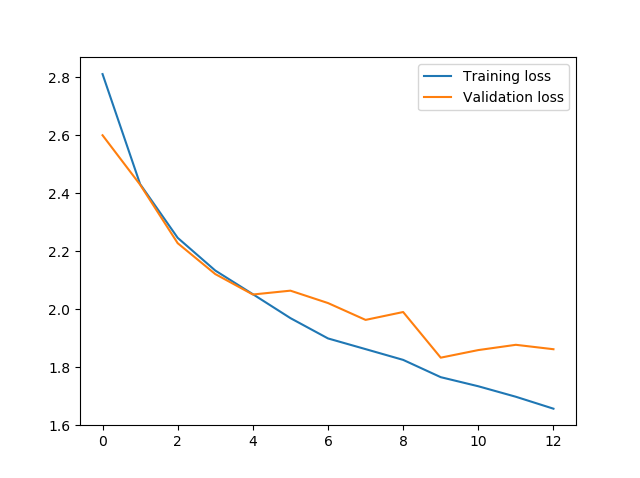
Estado final tras los experimentos

La primera transformación que se incluye son volteos verticales y horizontales, pues no cambia la clase de los objetos que estamos tratando (CIFAR-100 contiene mayoritariamente animales, vehículos y objetos caseros):





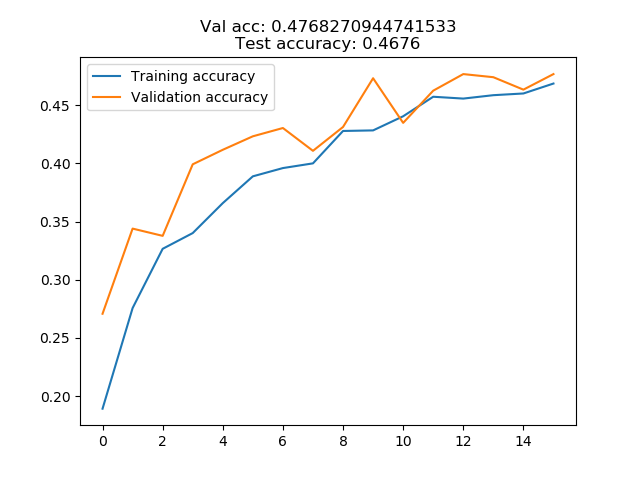
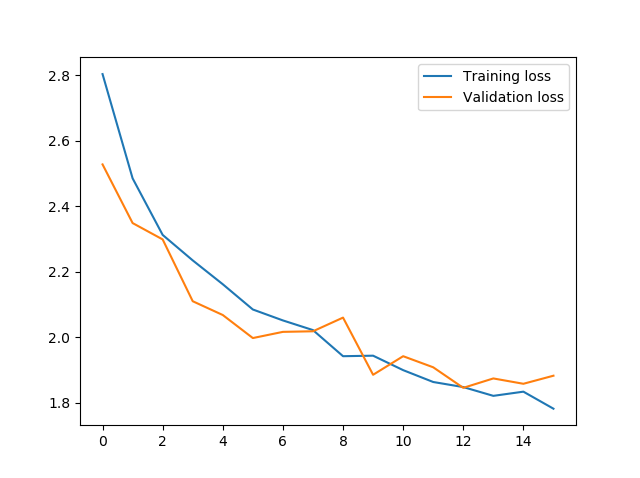
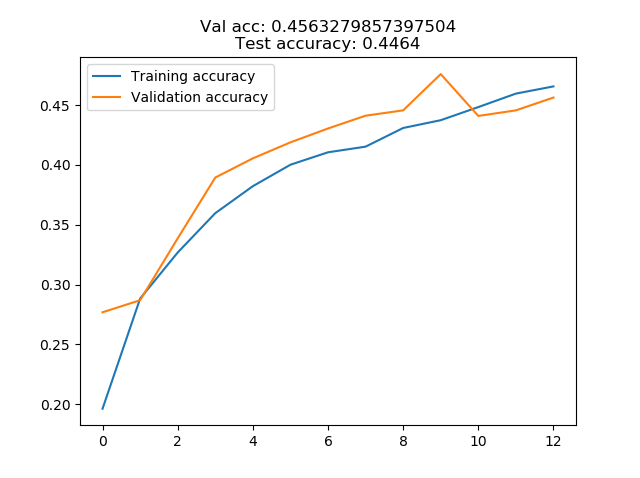
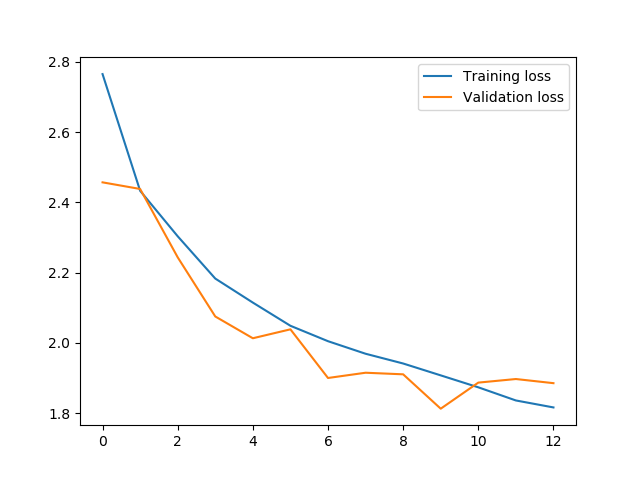
Vemos que no se consigue mejora y por tanto se prueba solo con los horizontales:



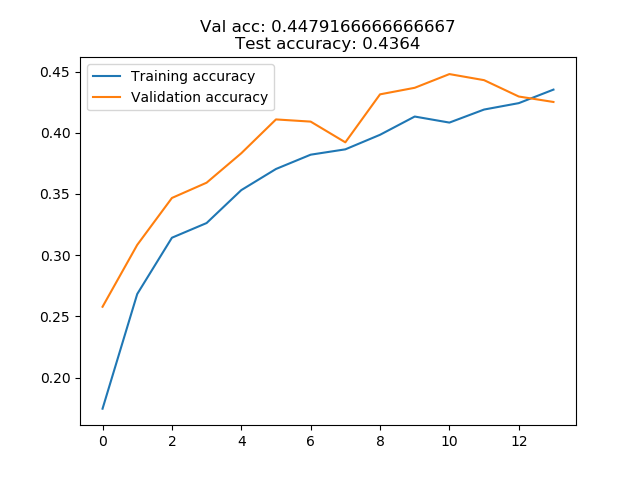
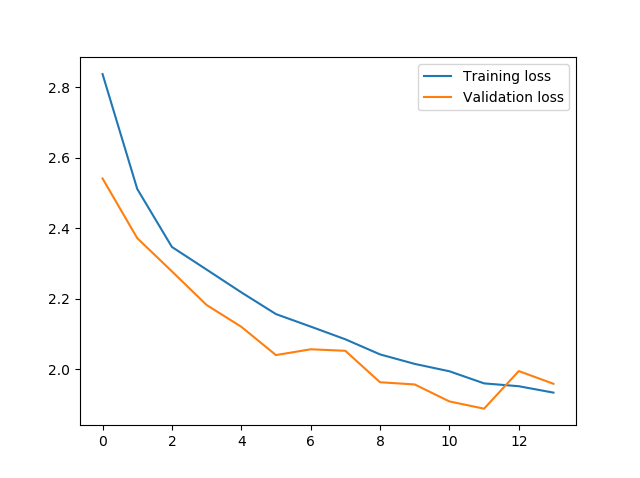
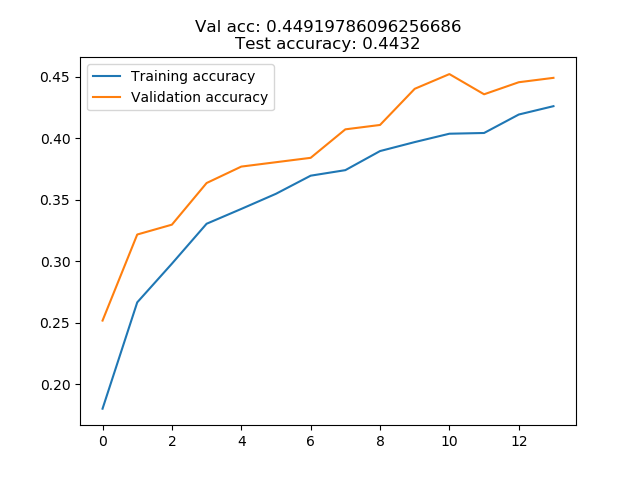
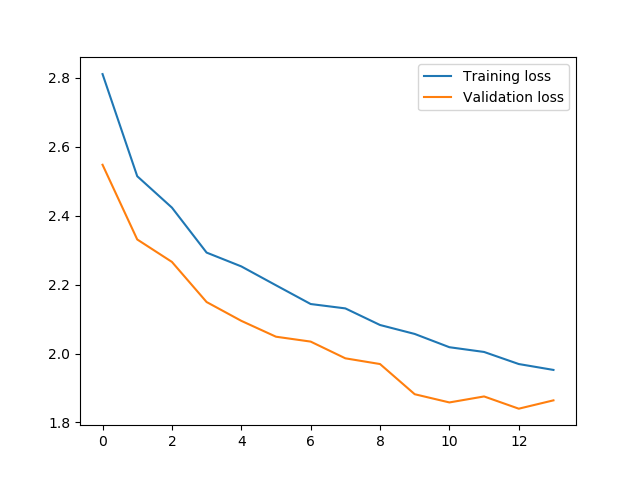
Ahora sí se aprecia una importante mejoría, y se mantiene la mejora.

Otra posible transformación es el zoom de la imagen. Esta aplicación tendría sentido en imágenes grandes donde el objeto a clasificar se encuentre centrado, en nuestro caso las imágenes son bastante pequeñas y no tenemos la certeza de la ubicación del objeto, así que es posible que esta mejora no nos sea de gran utilidad si se aplica en gran medida.

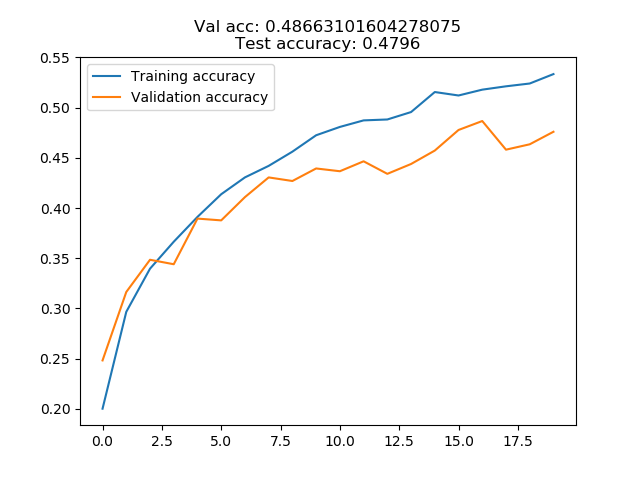
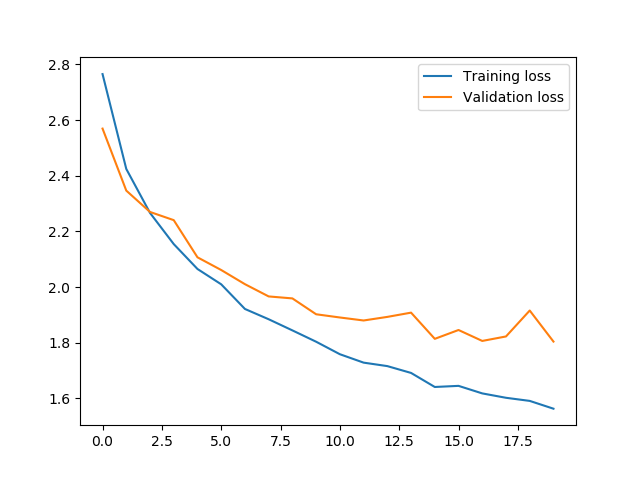
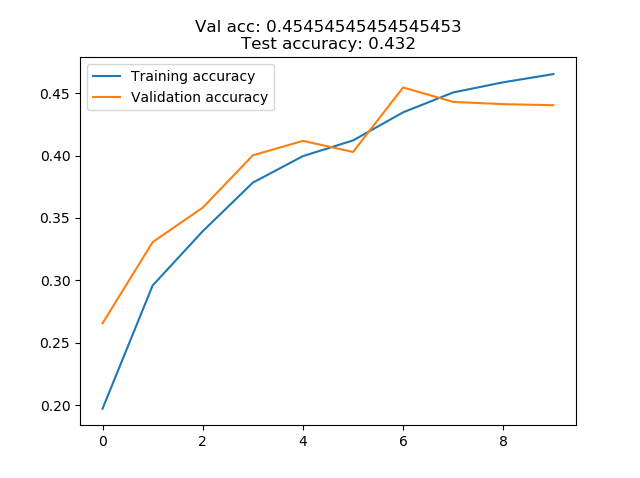
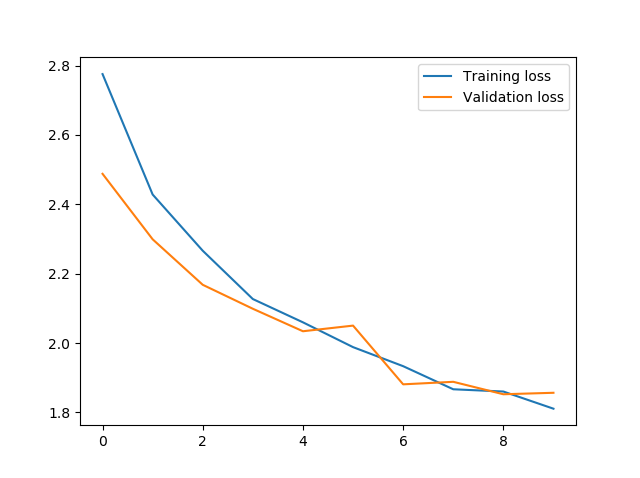
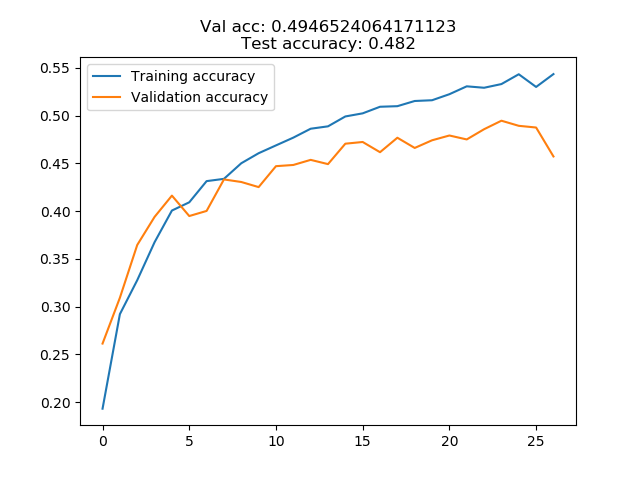
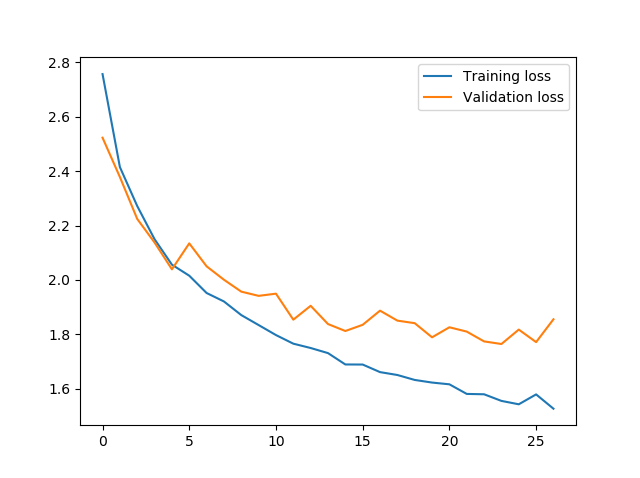
Para comprobarlo, se prueba con un valor pequeño (0.3):



No se nota mucha diferencia con los resultados anterior, se aumenta a 0.5:



Aquí se aprecia el deterioro de la calidad. Se prueba una última vez con un valor muy pequeño 0.15:



Y ya vemos que con esta probabilidad de zoom la mejora si es significativa y se decide proseguir con este estado de la red, y un nuevo *accuracy* en media de: **0.478**

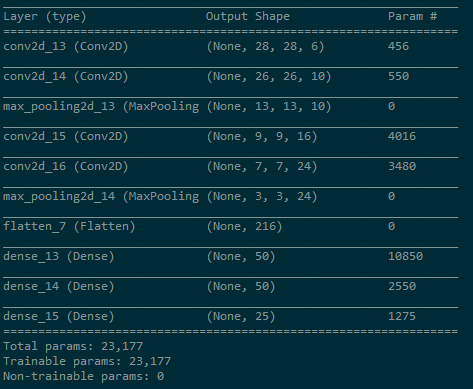
Ampliación de la arquitectura

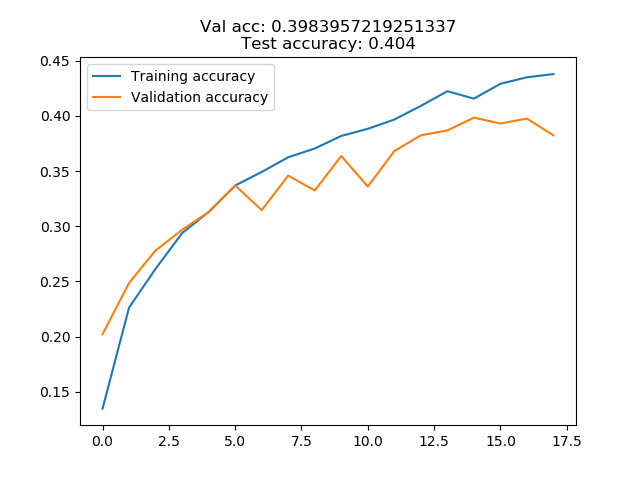
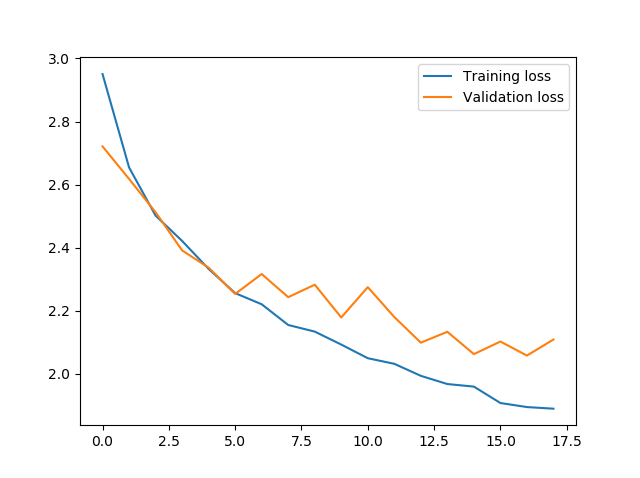
Una red más profunda es capaz de aprender problemas más complicados, ya que aumentamos la clase de funciones sobre la que trabaja.

Esto puede ser un problema cuando la red ya era capaz de por sí de aprender el conjunto de entrenamiento, y es que al profundizarla tiende en mayor medida a realizar *overfitting*. Aunque por otro lado una red más profunda puede extraer mejores características y clasificar con mayor facilidad.

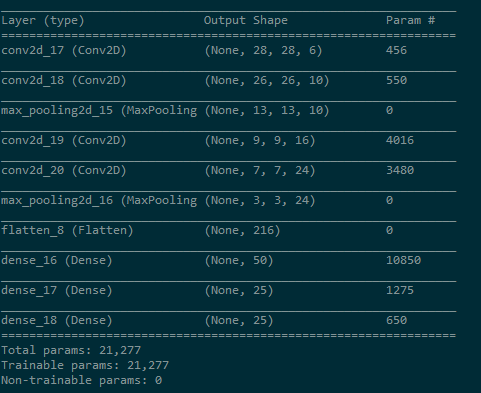
Se prueba primeramente añadiendo más capas convolucionales y completamente conectadas:

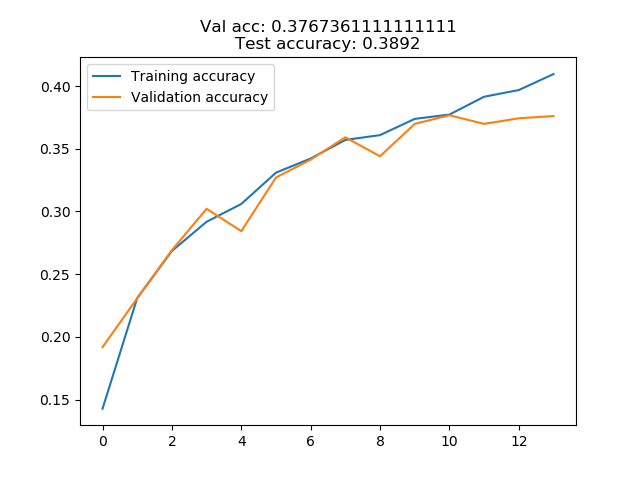
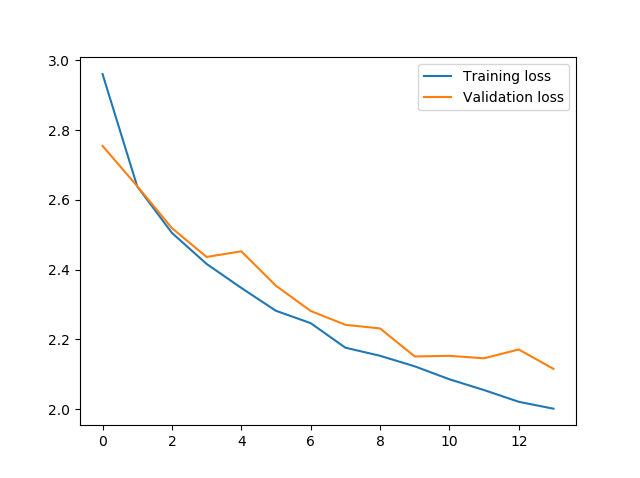
1ª prueba





2ª prueba

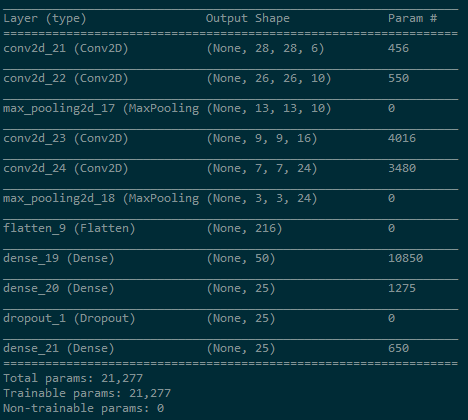


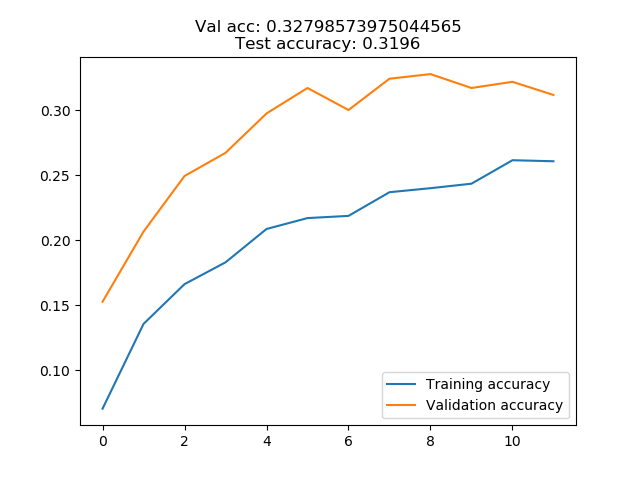
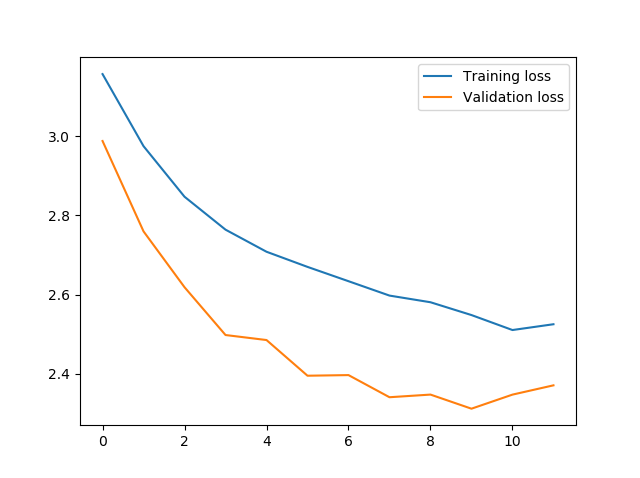


Consiguiendo un empeoramiento claro.

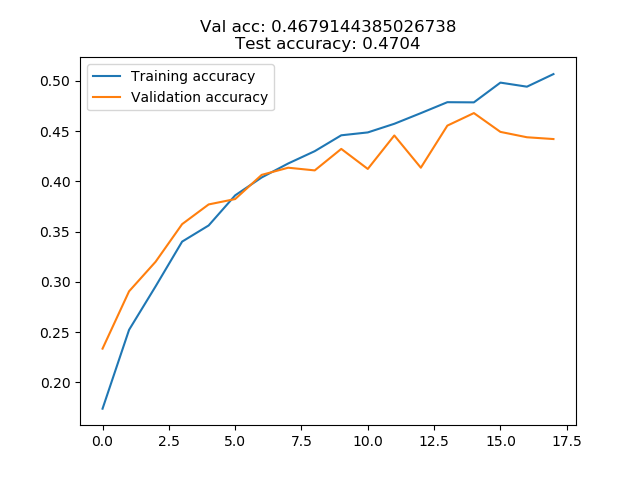
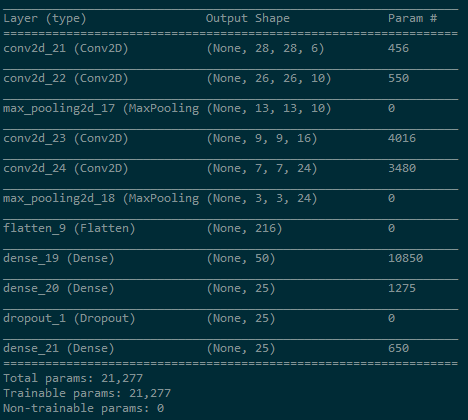
Se añaden regularización mediante Dropout:

1ª prueba



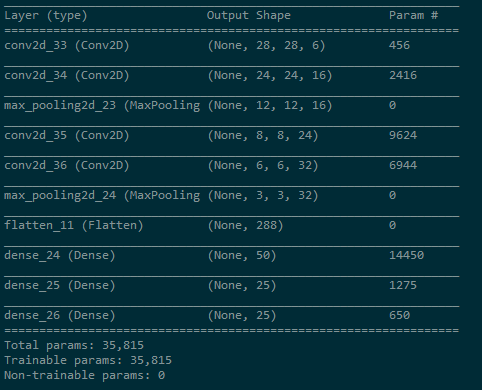


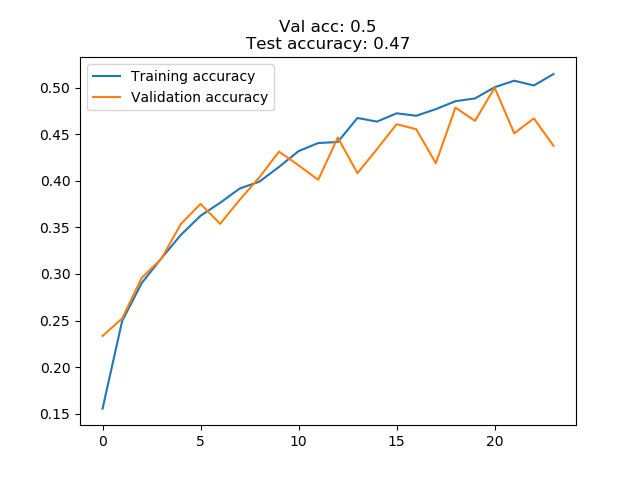
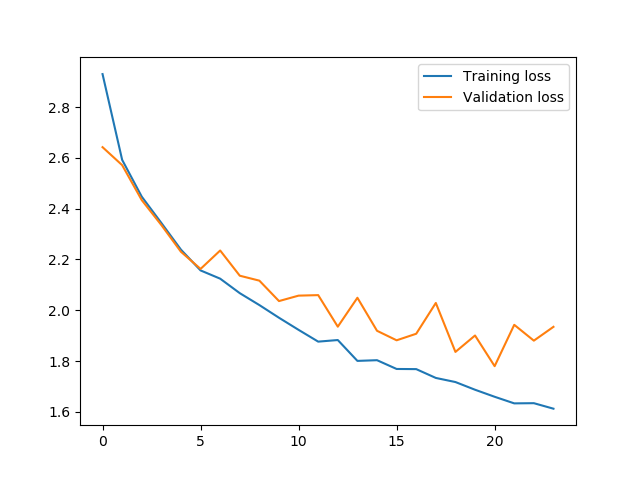
2ª prueba



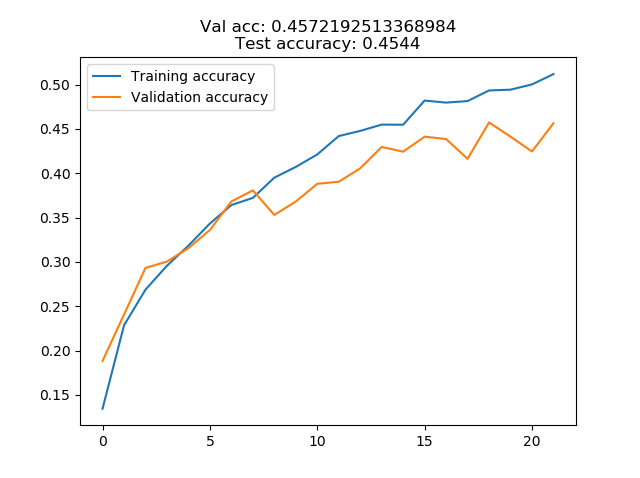
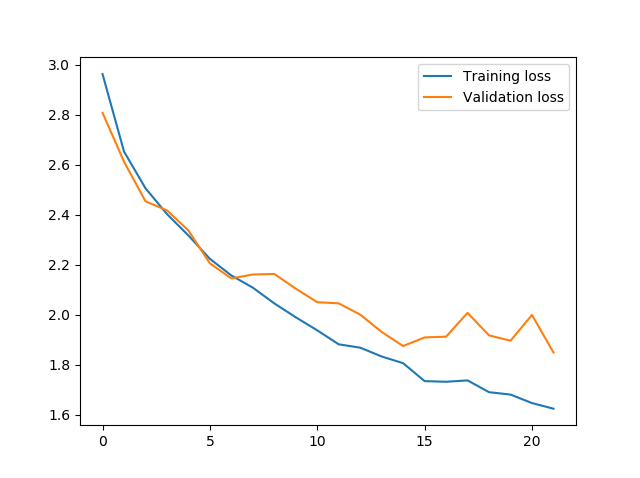
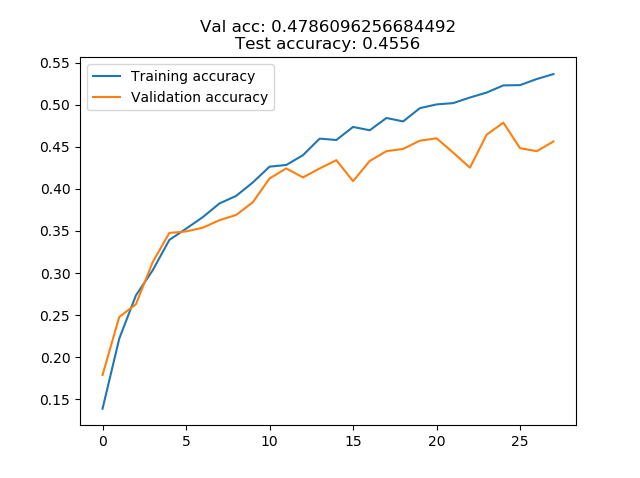
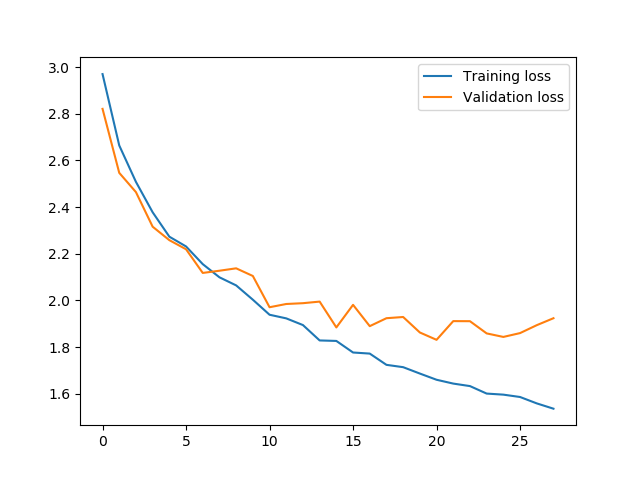
Sin conseguir mejora. Las gráficas de entrenamiento y validación están juntas, indicando que el problema no está en la generalización, sino en la capacidad de aprendizaje de la red.

Se vuelve a experimentar con diferentes tamaños de máscara en la convolución:





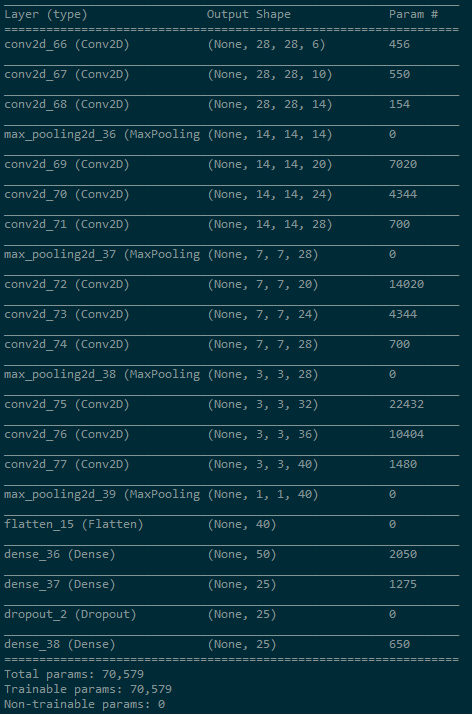
Y aquí sí parece que mejore la red, se experimenta más con ella:

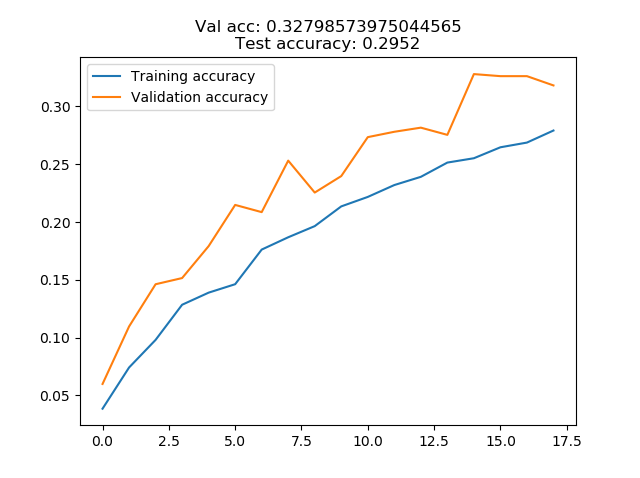
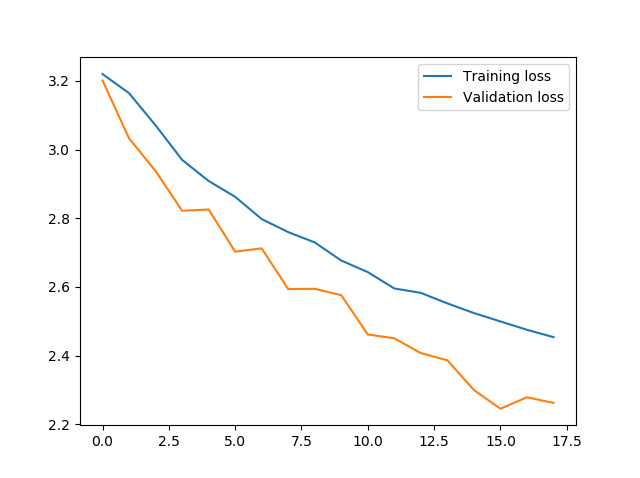


La gráfica muestra que tiene bastante variabilidad y parece que ese *accuracy* fue un golpe de suerte, quizás ajustando los hiperparámetros se podría volver a lograr ese 50%.

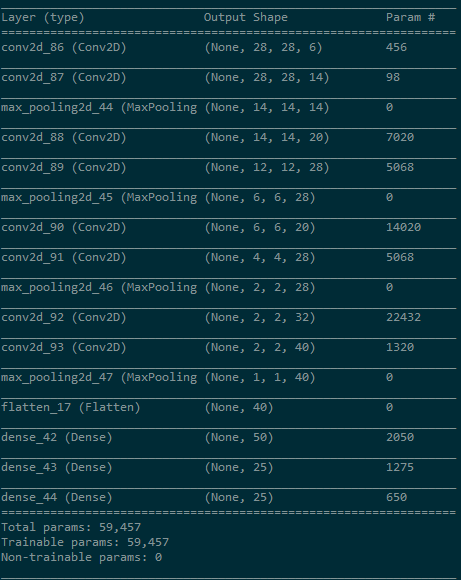
Se decide por tanto experimentar con otras redes, en este caso más profundas:

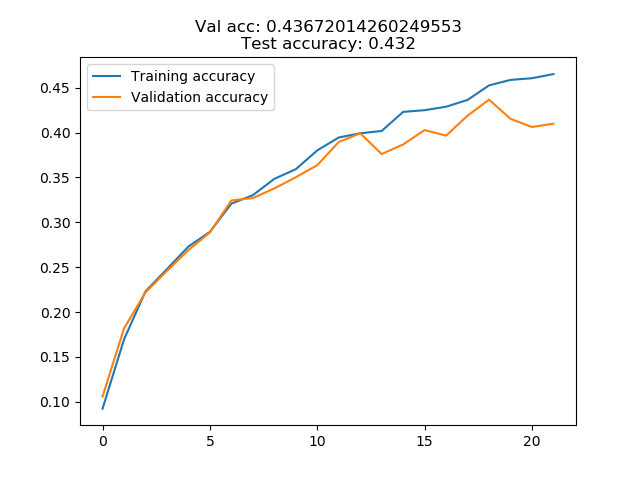
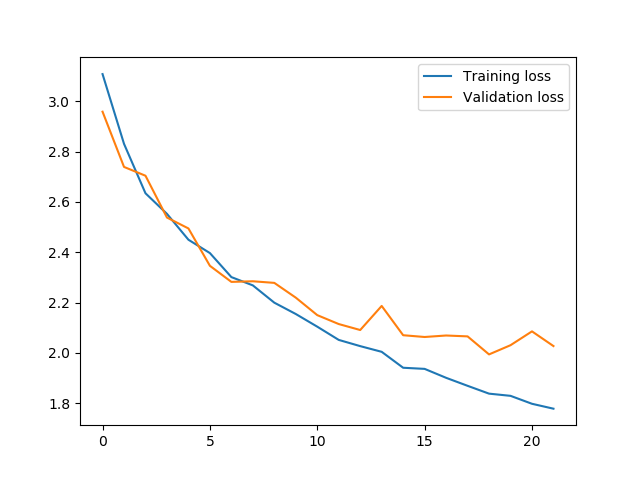
1ª prueba



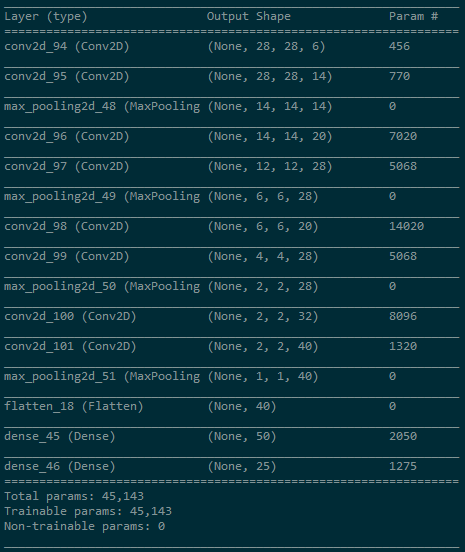


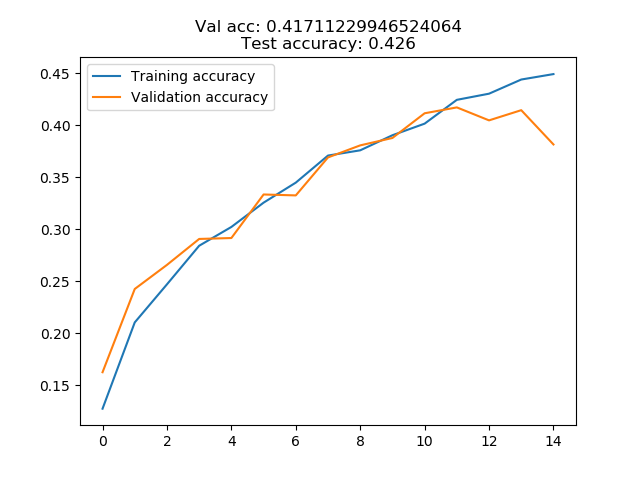
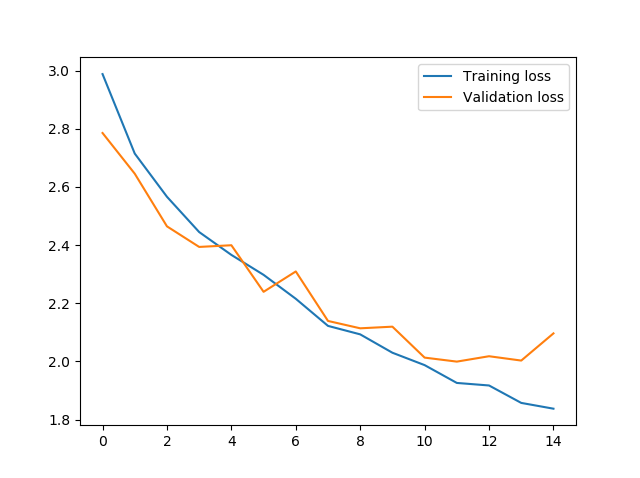
2ª prueba





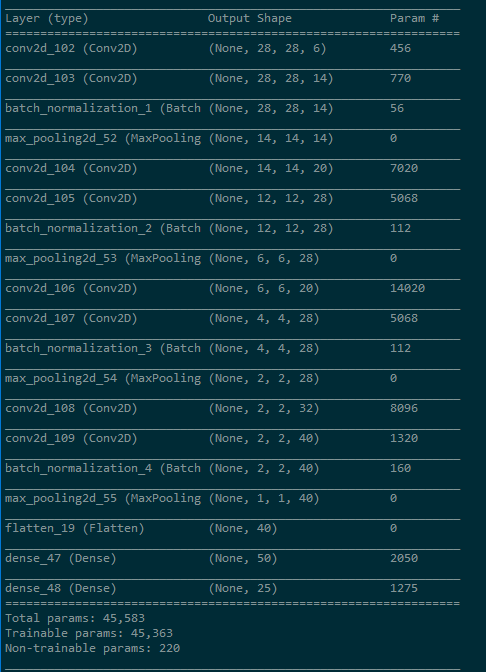
3ª prueba

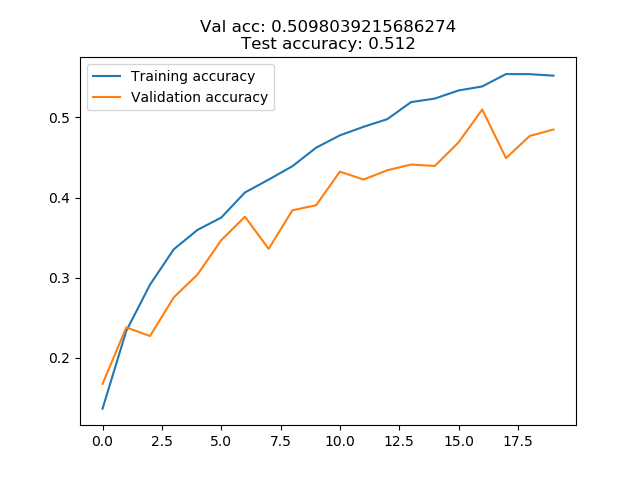
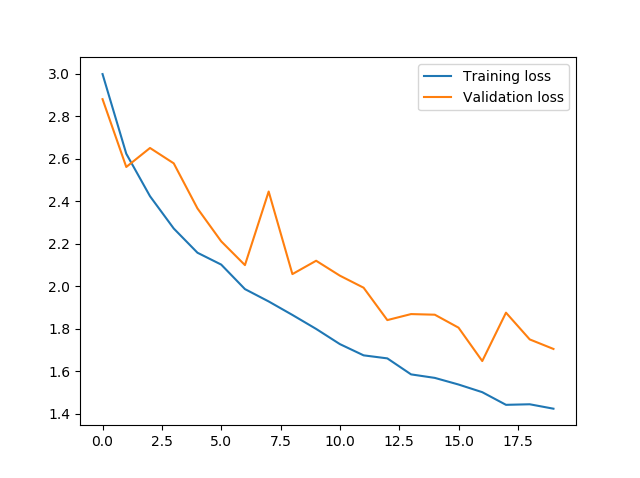
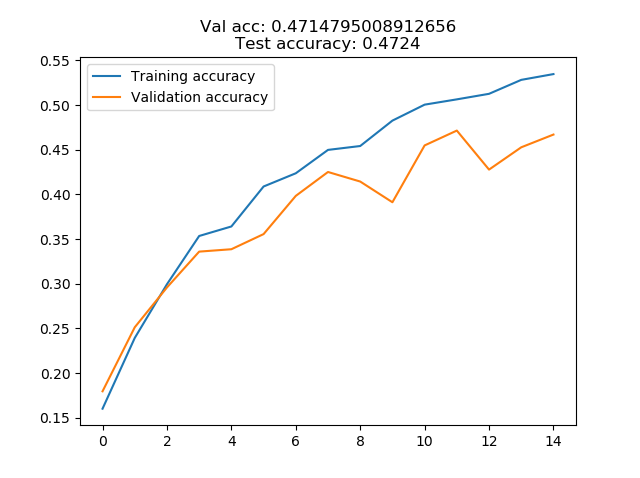
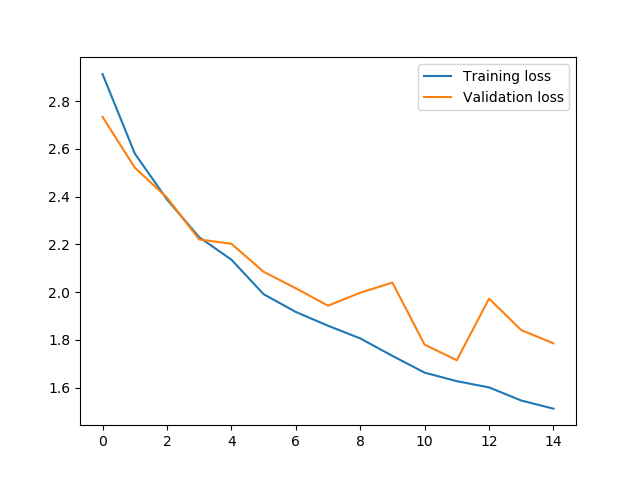
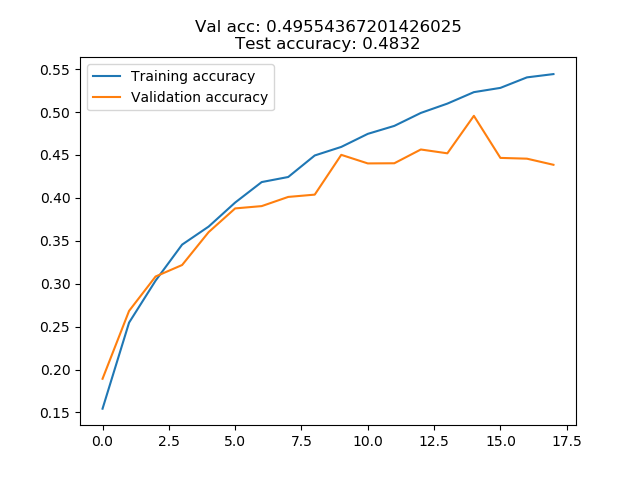
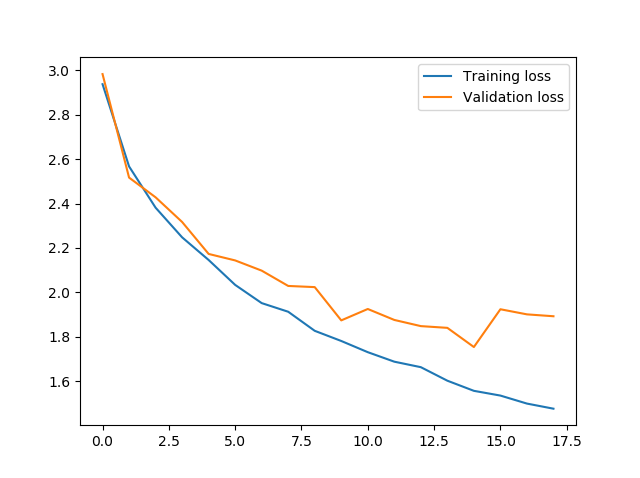




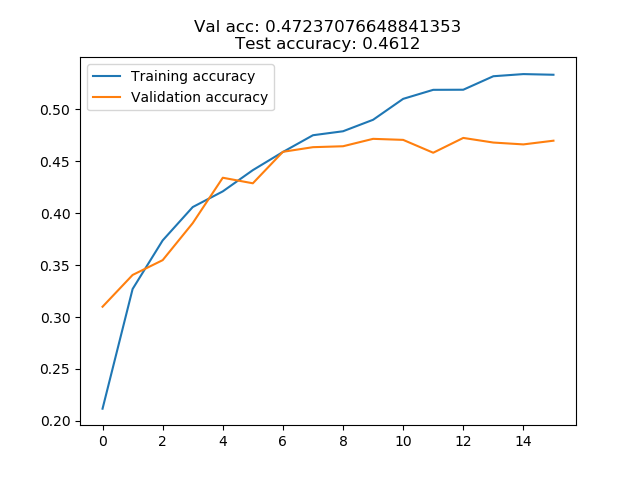
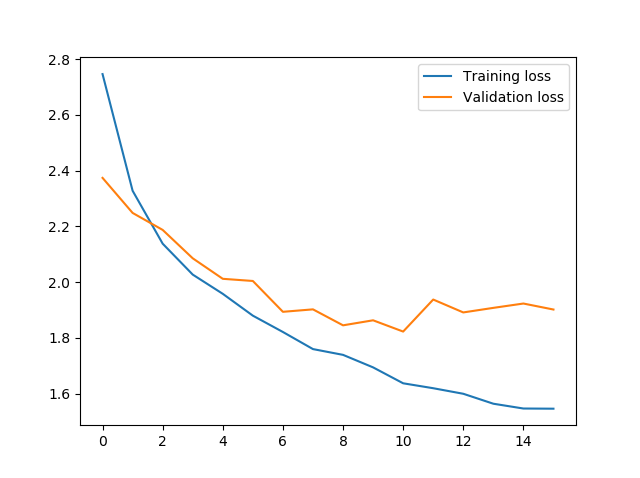
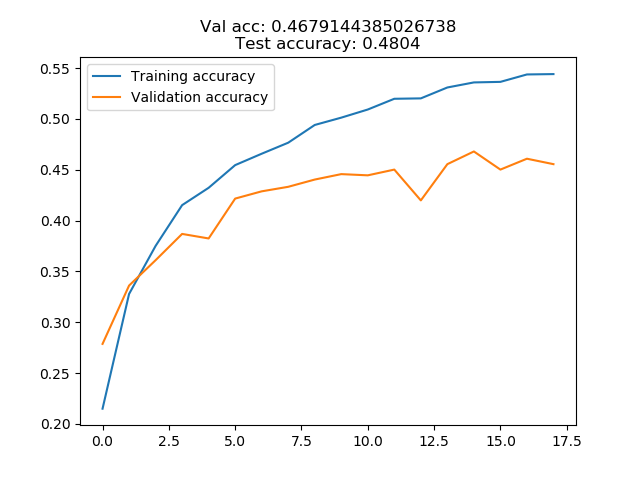
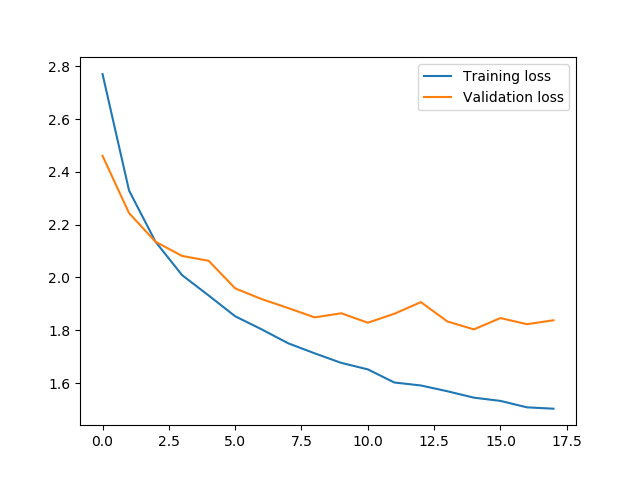
Pero no se aprecia mejoría. Un posible motivo es que tras tantas capas de convolución la falta de normalización haga que la red no mejore. La normalización inicial se puede acabar perdiendo tras varias capas de convolución, ya que estas no aseguran que se mantengan los rangos de los datos.

Se vuelve a experimentar añadiendo *BatchNormalization* por cada bloque convolucional:





Y vemos que el modelo sí mejora. Para comprobar que es la combinación de profundidad y normalización la que hace que el modelo funcione mejor, se añade a la versión no profunda *BatchNormalization,* obteniendo:



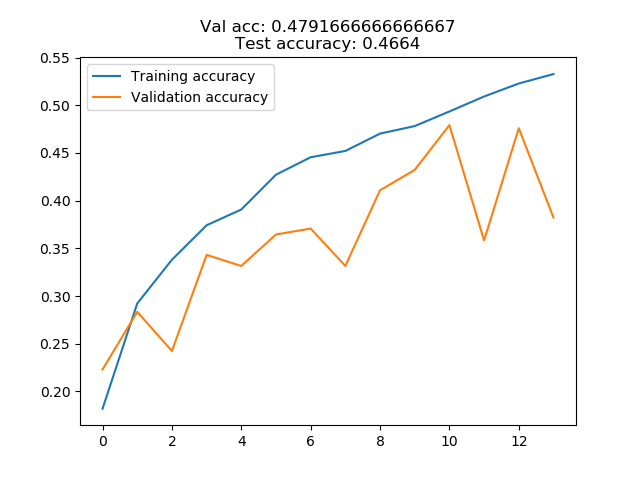
Y como no se alcanzan los valores de la red anterior nos quedamos con el nuevo modelo y una *accuracy* de: **0.488**

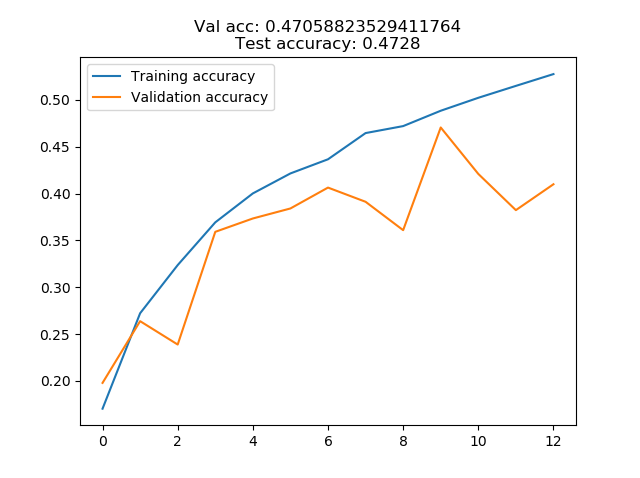
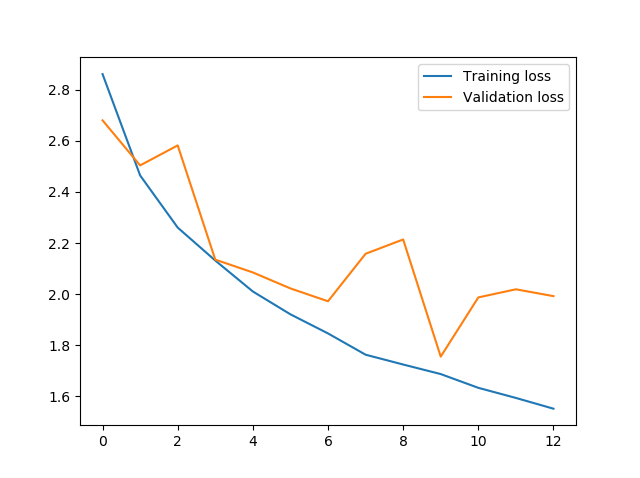
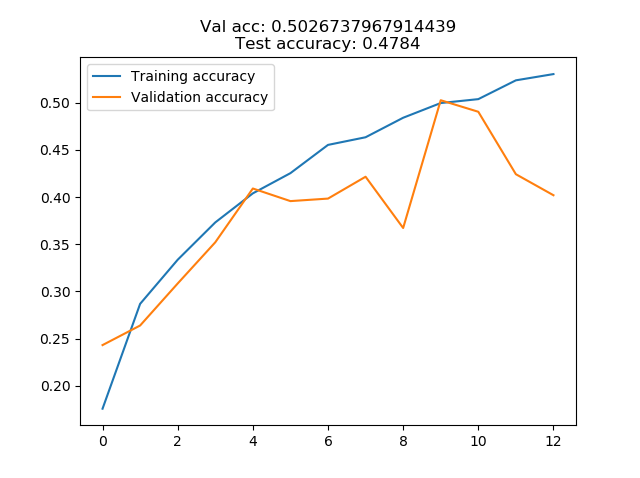
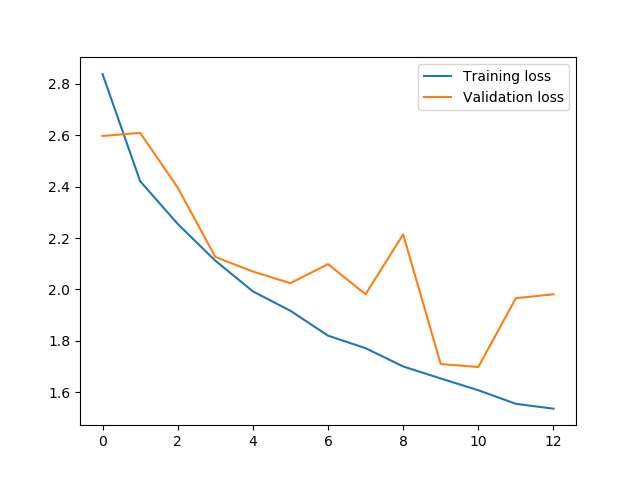
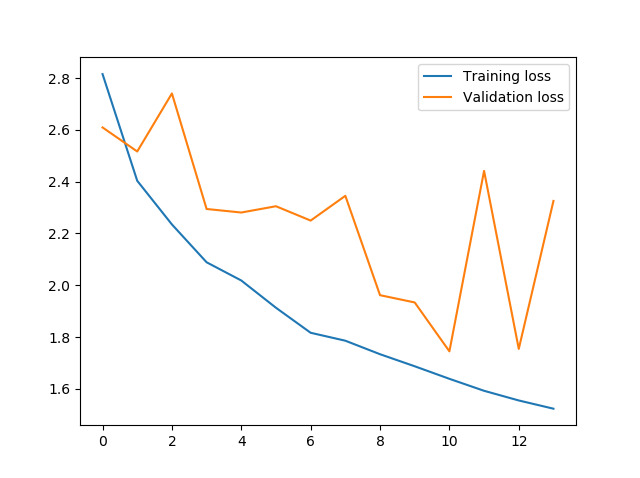
Normalización por capas

Puesto que se normaliza al principio, resulta apropiado volver a hacerlo dentro de la red, pues al aplicar las capas de Conv2D y Dense no se asegura que se mantenga la normalización.

Como ya se ha experimentado con *BatchNormalization* previamente, probamos a alterar la posición de las capas.

Añadiéndolas antes de las capas ReLU.





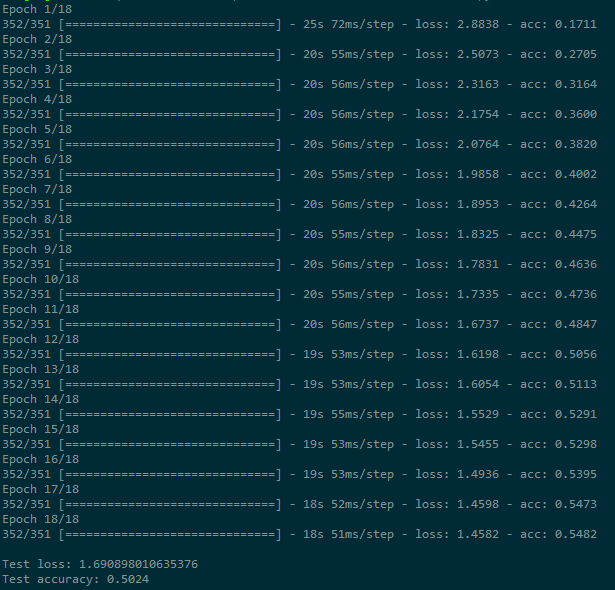
Se aprecia una mayor inestabilidad en la accuracy de validación. Es posible que se esté perdiendo el efecto de la normalización tras aplicar la capa ReLU.

Comparando las medias la diferencia es mínima, y por tanto se ve más sensato optar por dejarla tras la capa ReLU

Modelo final

Con todos los cambios hechos, se enfrenta al modelo con los datos de test, y así ver su calidad con datos nuevos. Para ello juntamos el conjunto de validación con el de entrenamiento y hacemos un entrenamiento final.

Los hiperparámetros elegidos son con mucha seguridad no óptimos, pero nos sirve para estimar la calidad.

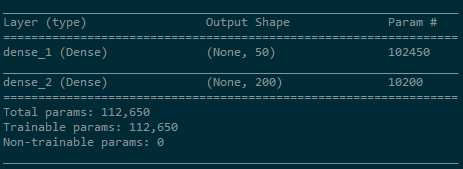


Apartado 3

Extracción de características

Se procede a construir el modelo ResNet50, introduciendo una capa GlobalAveragePooling a la salida (quitándole previamente el clasificador), de manera que se obtenga una salida de una sola dimensión.

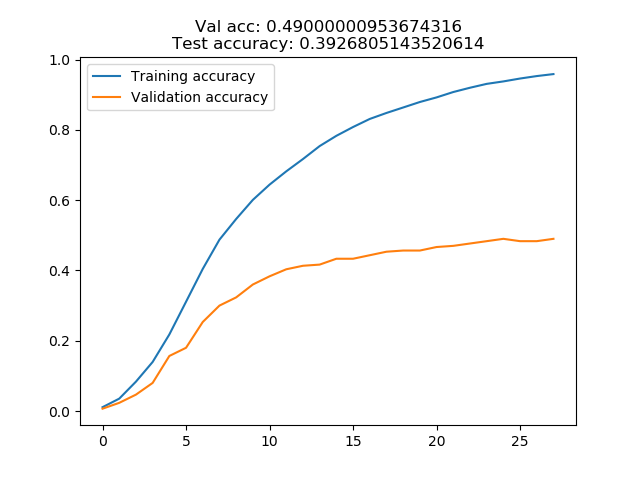
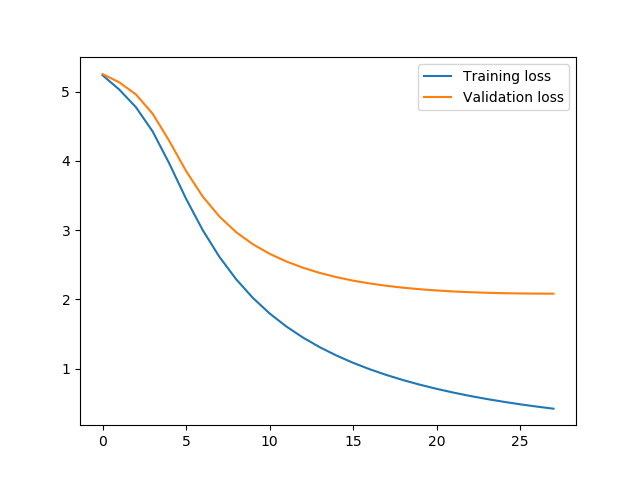
Posteriormente, construimos un clasificador formado por dos capas completamente conectadas para que, a partir de las características previamente extraídas de ResNet, clasifique las imágenes en nuestras 200 clases.



Se elige un tamaño de *batch* inferior (16) pues ahora contamos con menos datos que antes.

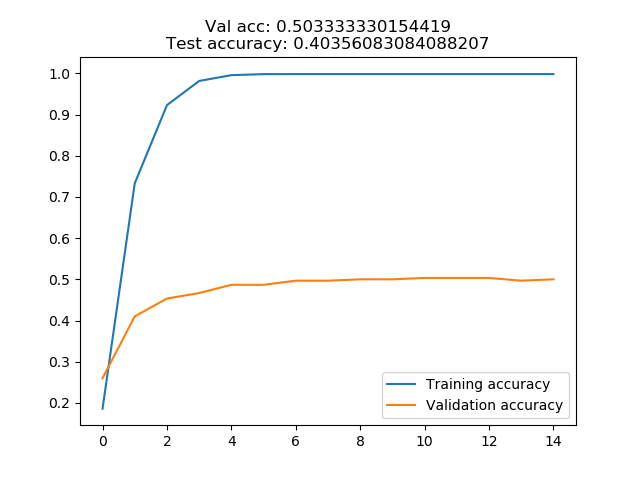
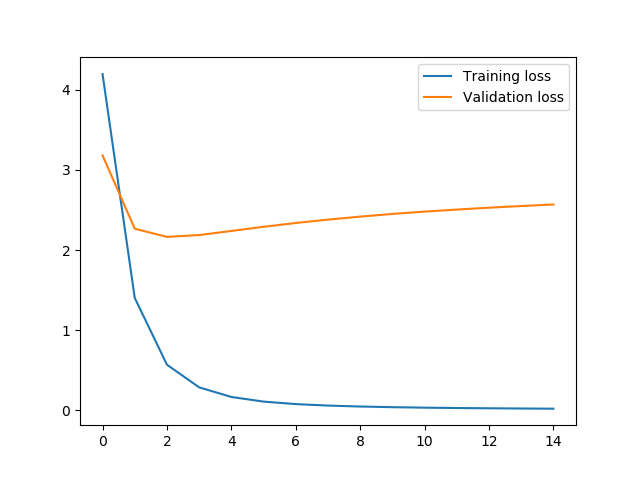
Pese a que en el apartado anterior se ha visto que RSMprop parece funcionar mejor que SGD para nuestro caso, se vuelve a comprobar para ver si era un caso concreto de nuestros datos.

Con parámetros por defecto:

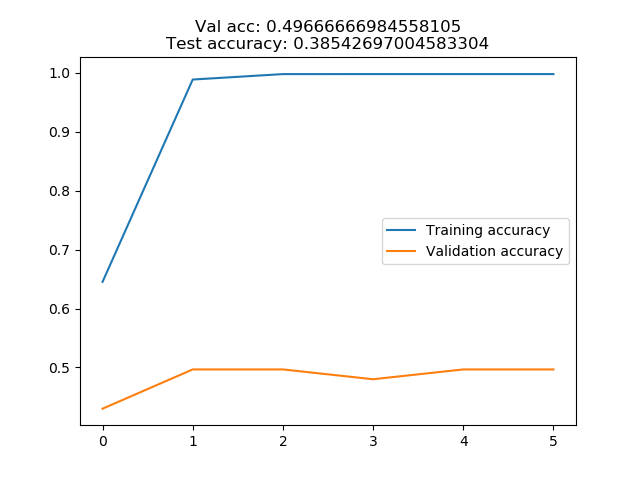
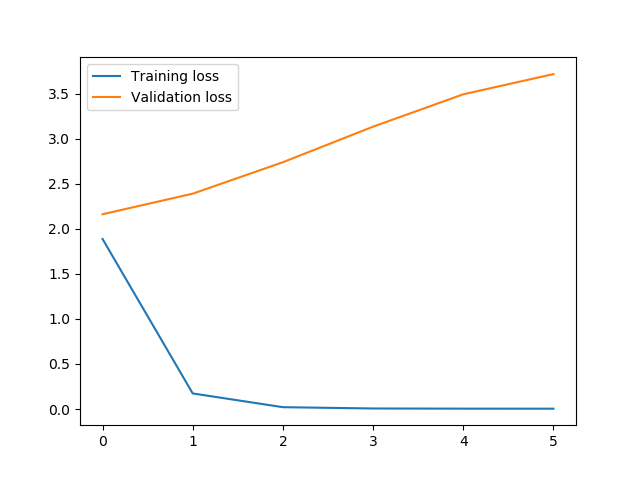


Con los parámetros de las diapositivas:



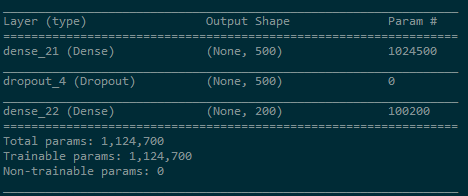


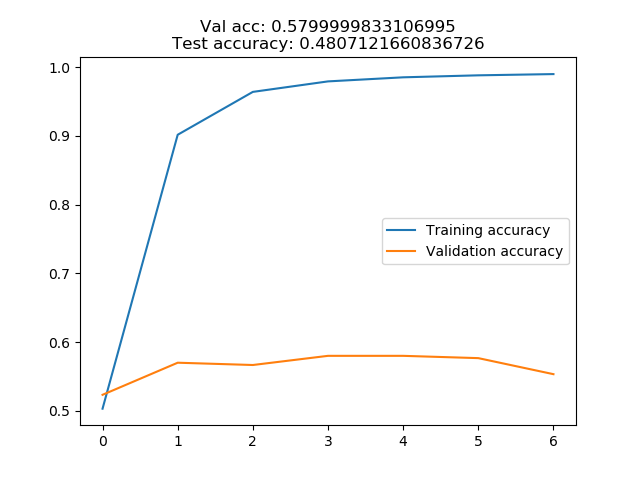
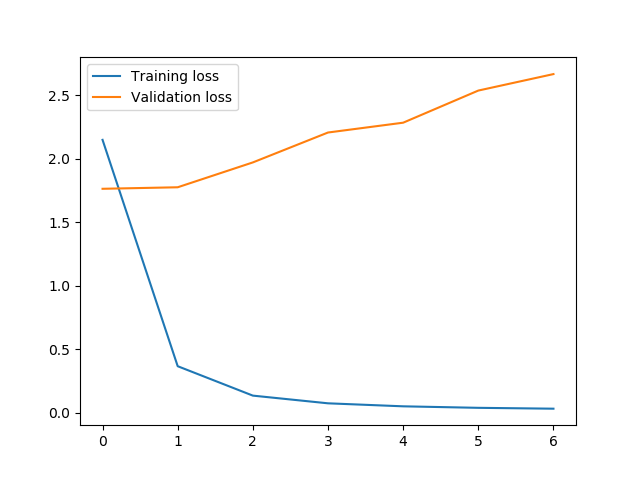
Y para contrastar, RMSprop por defecto:



Se sigue con RMSprop, a pesar de tener una peor *accuracy* (aunque por poco margen) con la esperanza de al optimizar su *learning-rate* podamos conseguir una curva de aprendizaje similar a la de SGD y con mejores resultados.

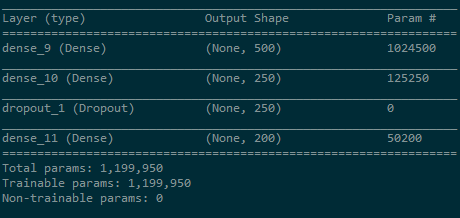
Vemos que el modelo sobreaprende en tan solo una época, por lo que buscamos mayor generalización. Para ello se define un nuevo clasificador añadiendo regularización con Dropout:

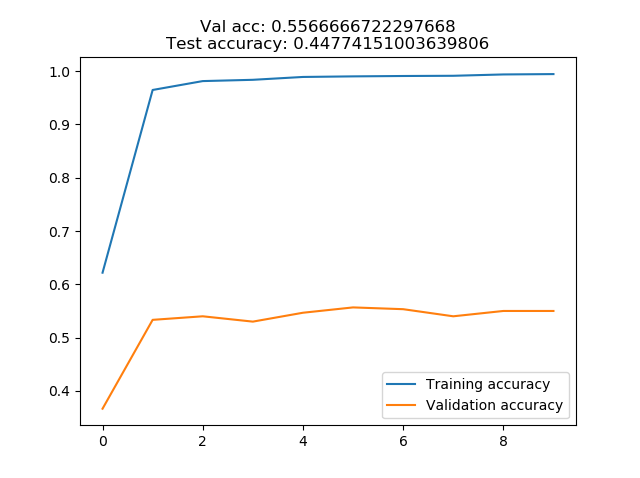
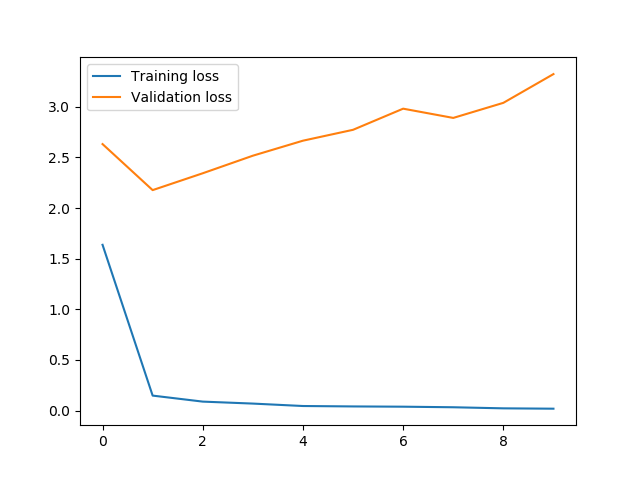




Aunque sube, el *accuracy* del modelo sigue siendo bajo para estar usando una red pre-entrenada. Tampoco resulta extraño viendo que solo contamos con 3000 datos de entrenamiento.

Probamos a modificar la arquitectura del clasificador:



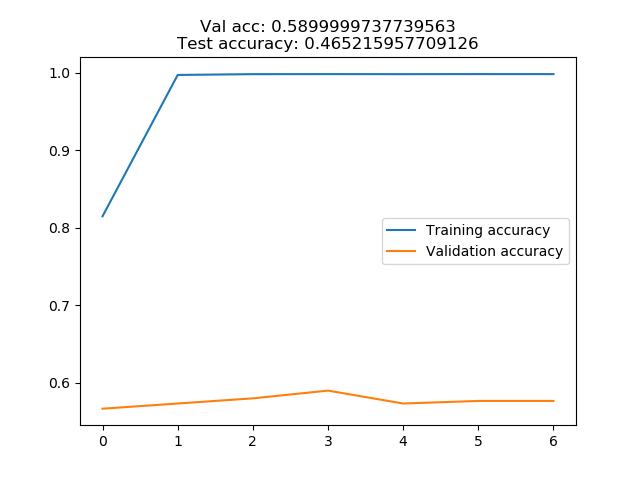
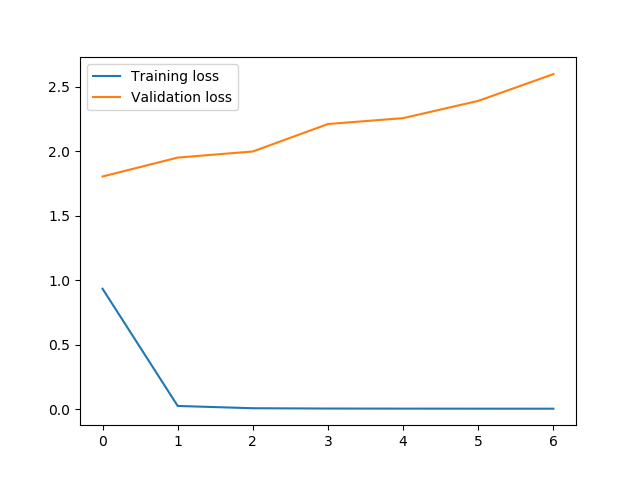
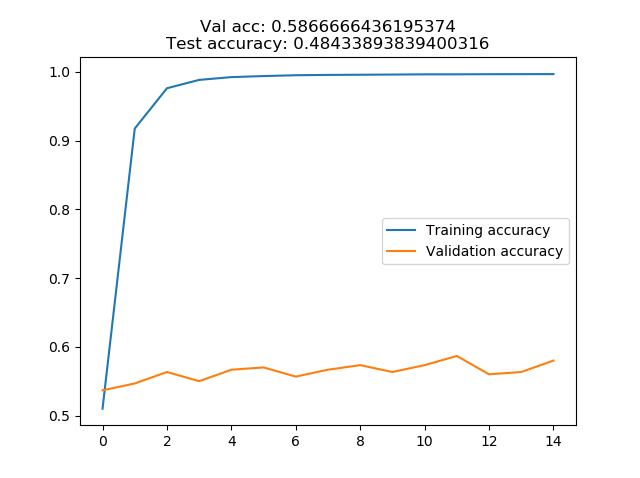
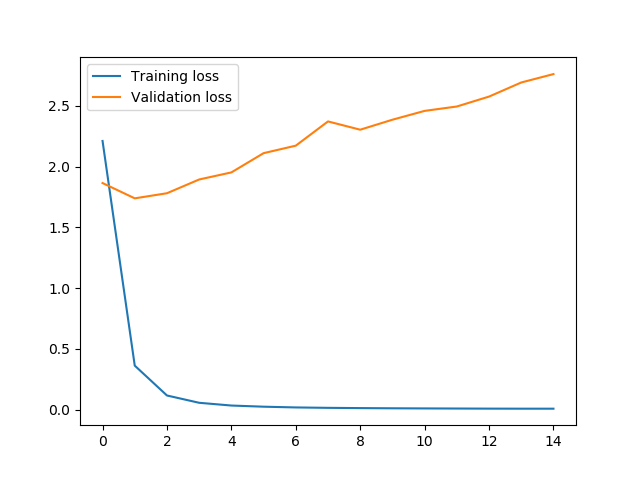


Realiza *overfitting* aún más rápido, demostrándonos que la arquitectura de por sí ya es suficiente para aprender el conjunto de entrenamiento, con la pega de que al ser tan potente aprende demasiado rápido y no generaliza.

Por tanto, se mantiene la arquitectura anterior.

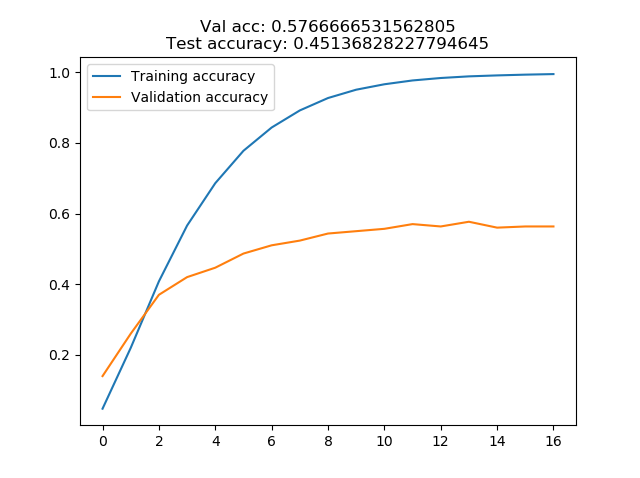
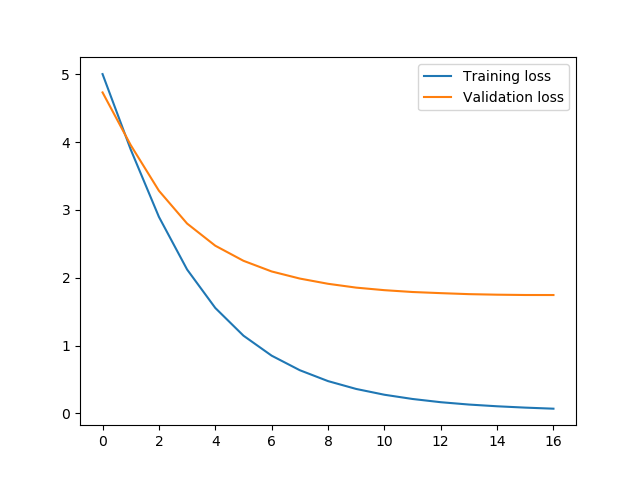
Para evitar que el error de training suba demasiado rápido, se decrementa el *learning-rate*:





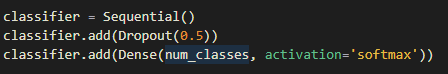
Y viendo que la curva mejora, se baja aún más la tasa de aprendizaje:

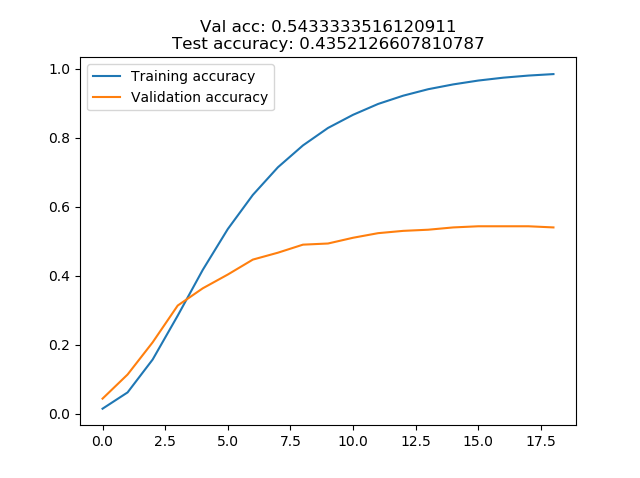
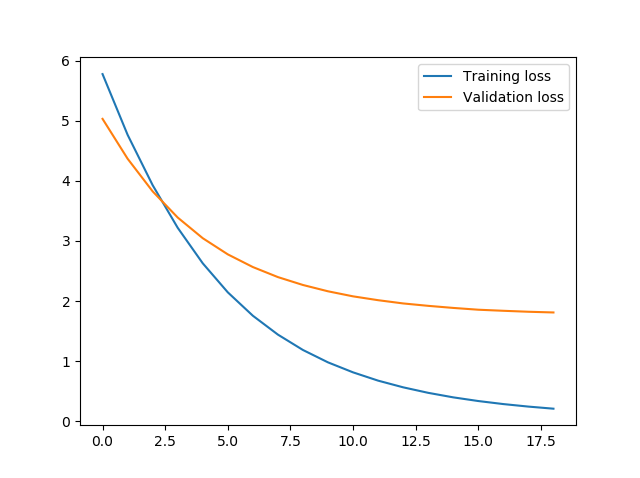




Y aunque conseguimos una curva de aprendizaje más aceptable, perdemos un 1% en la *accuracy* del modelo.

Probamos a simplificar la arquitectura del clasificador, pues ya hemos visto que la red en sí es demasiado potente.

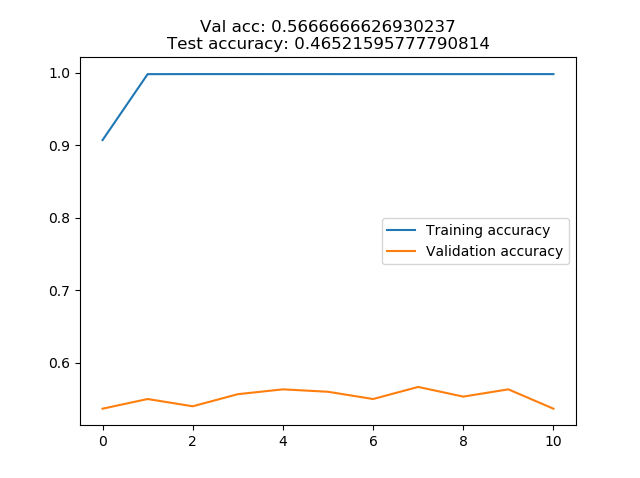
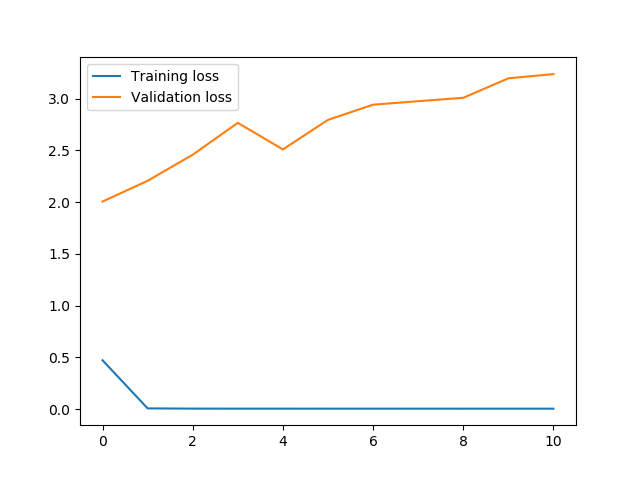




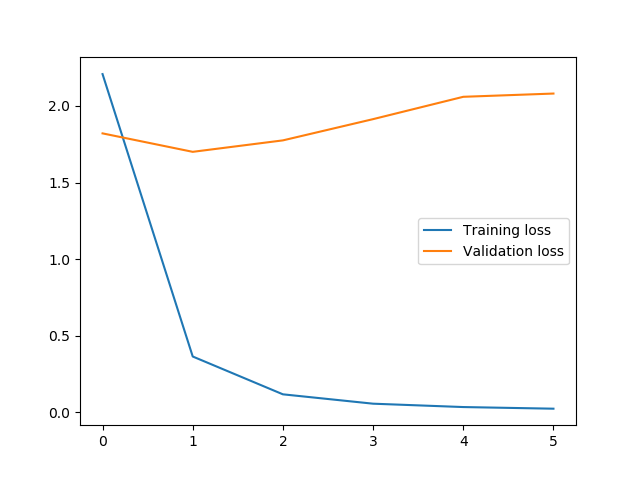
Pero notamos un bajón en el *accuracy*, esto nos dice que no es el clasificador el que es demasiado potente para nuestros datos, sino la red.

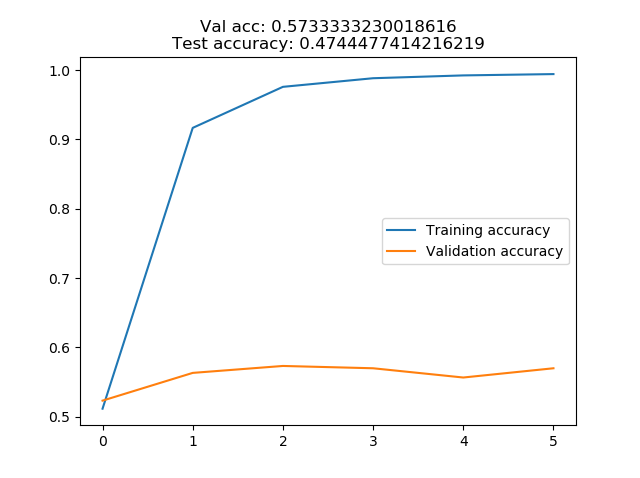
Intentamos por tanto optimizar los hiperparámetros de nuestro mejor modelo para poder obtener el mejor resultado posible (se considera que lo óptimo habría sido una búsqueda en grid automática):

* Bajamos el tamaño de batch a 8.

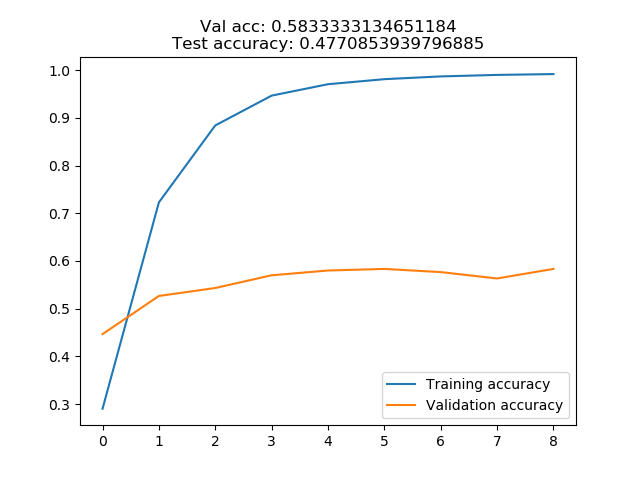
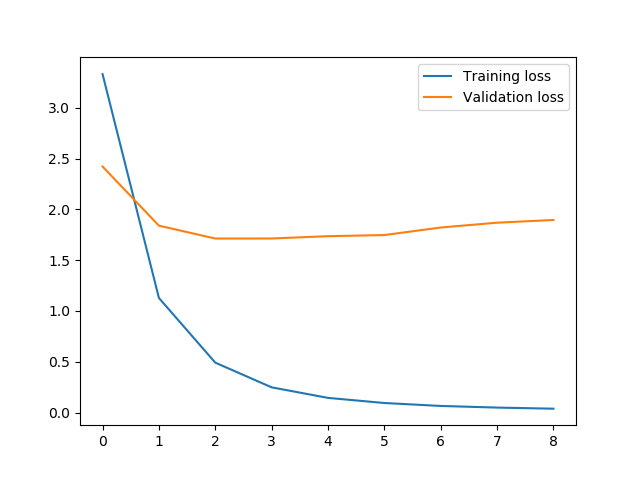


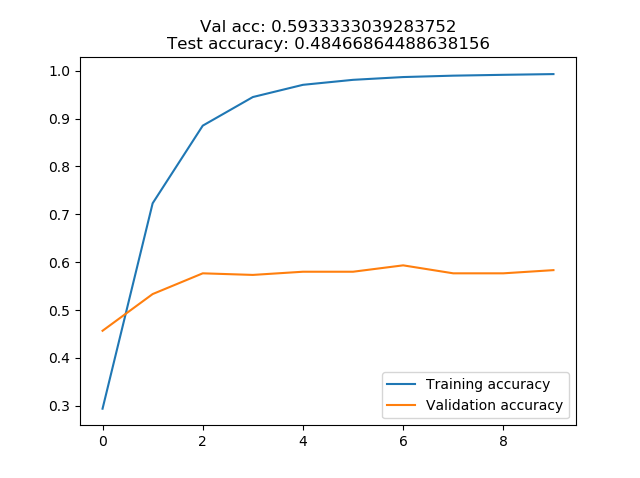
* Se aumenta la cantidad de Dropout a 0.8.

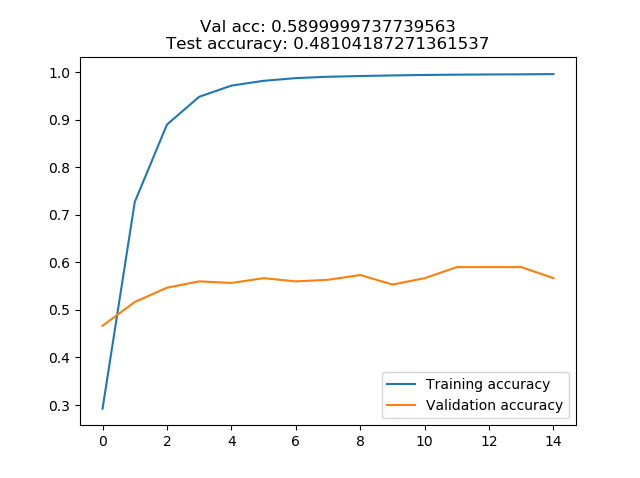
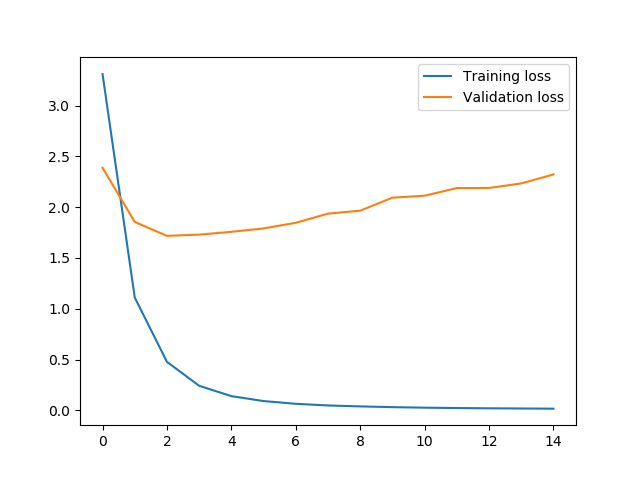
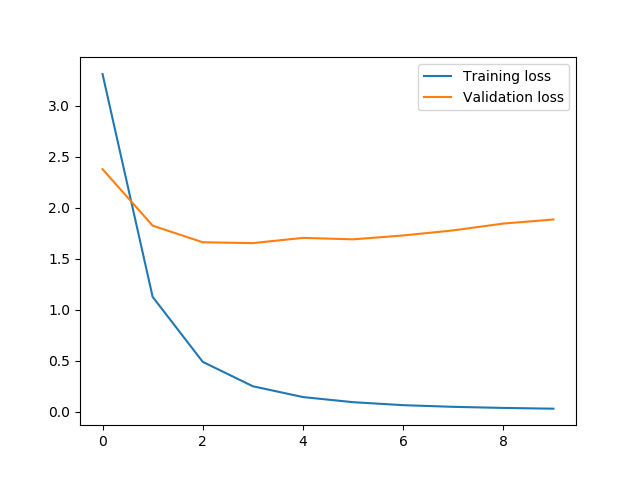




* Dejando el Dropout aumentado, se incrementa el tamaño de batch a 32.







Este último modelo es el que nos da mejores resultados de todos.

Para un modelo final la *accuracy* obtenida es bastante baja, pero parece que no es posible mejorarla en gran cantidad usando el conjunto de datos que se nos proporciona.

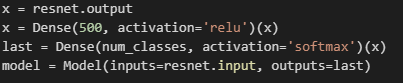
Fine-tunning

En este caso cambiamos las capas superiores de ResNet50 para adaptarlas a nuestro problema (con un nuevo clasificador), y reentrenamos la red completamente con nuestros datos.

Se esperan dos posibles resultados:

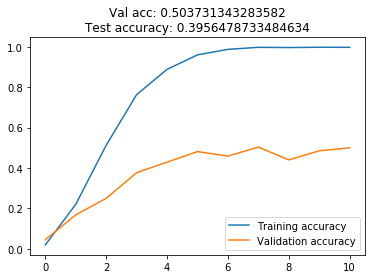
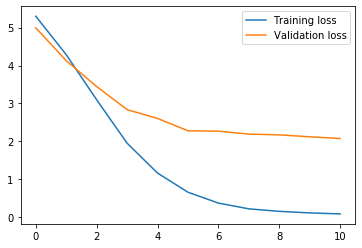
* Que el adaptar todos los pesos de la red haga que resuelva de mejor manera nuestro problema.
* Que los pesos de ImageNet sean mejores que los cambios que puedan forzar nuestros datos, pues contamos con muy pocos. Para evitar esto una buena vía podría ser utilizar un *learning-rate* muy bajo y con cierto *decay* para ajustar lo mínimo necesario.

En principio comenzamos con dos capas completamente conectadas:

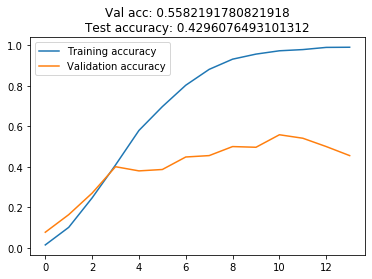


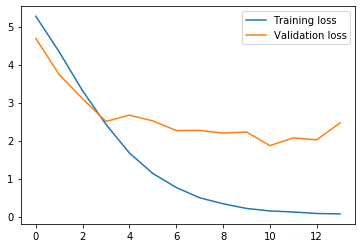
Contamos con pocos datos de entrenamiento (3000) por lo que se sigue dejando un tamaño de *batch* pequeño (16)

Una primera prueba con SGD nos da una estimación de la capacidad del modelo:

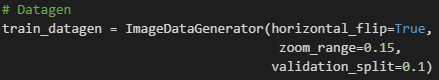


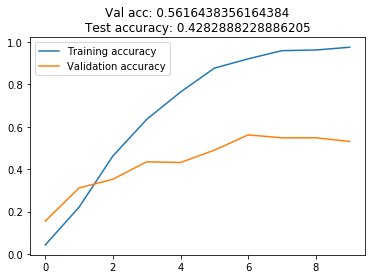
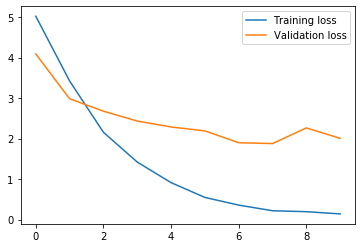
Añadimos regularización con Dropout, buscando una mayor generalización:



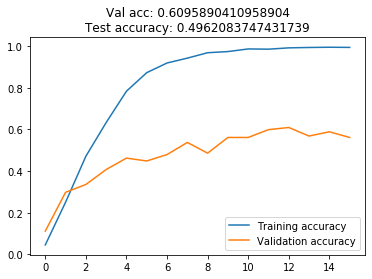
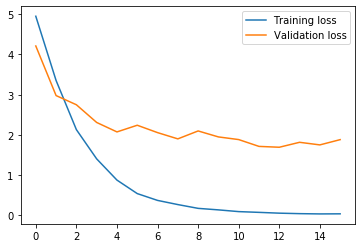


Seguimos intentando generalizar, y para ello añadimos *data-augmentation*. Concretamente, los parámetros elegidos con BaseNet:

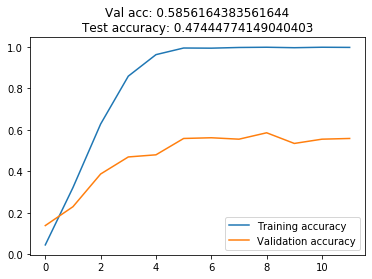
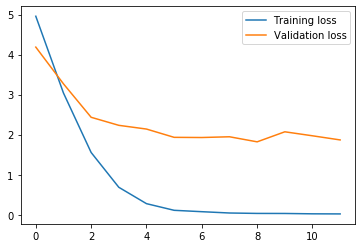




Ajustando hiperparámetros, bajamos el tamaño del *batch* a 8.

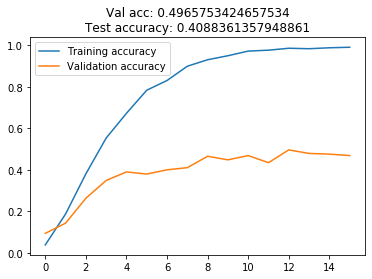
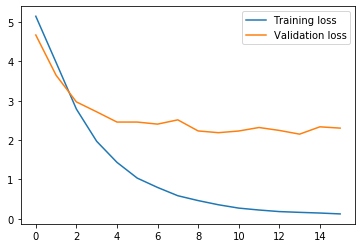


Cambiamos de optimizador y probamos con RMSprop:

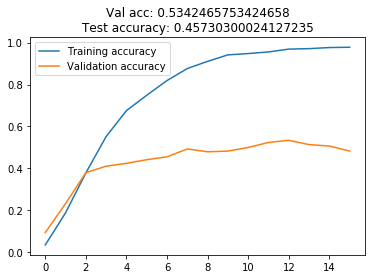
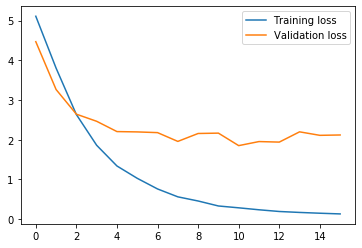


Las primeras capas de la red, estando entrenadas con ImageNet, son capaces de sacar características generales de las imágenes y puede no ser necesario entrenarlas de nuevo. Un ajuste de los pesos de las últimas capas podría dar mejores resultados al enfocarlos hacia nuestro modelo.

Siguiendo esta idea se congelan todas las capas de ResNet excepto las 10 últimas.



Con resultados bastante peores, se vuelve a probar, pero con 19 capas en este caso:



Y no nos da mejores resultados, esto nos indica que los pesos de ImageNet no nos son 100% útiles y el ajuste fino de la red entera es una mejor opción.